Sistemas Informáticos
Curso 2004-2005

Implementación del estándar DRMAA para enviar, gestionar y terminar trabajos

Patricia Arroyo Peceño
Sergio Espartero Díaz
Beatriz Palazuelos Moya

Dirigido por:
Prof. Ignacio Martín Llorente.
Dpto. Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

Índice

Presentación

Resumen del Proyecto

El objetivo de este proyecto es la implementación de la especificación DRMAA sobre la herramienta GridWay.
GridWay (www.gridway.org) surge como una herramienta para facilitar el uso de las tecnologías grid, permitiendo que el programador no se tenga que encargar de la planificación de tareas, el descubrimiento y la selección de recursos…
GridWay esta basado en Globus, permitiendo la ejecución de trabajos denominada 'enviar y olvidar ('submit and forget'). Permitiendo que el entorno de grid sea dinámico, ya que permite la migración de trabajos desde recursos más lentos o que hayan dejado de funcionar.
DRMAA (Distributed Resource Management Application API) es una especificación para el envío y control de trabajos en uno o más gestores de recursos distribuidos (DRM's), desarrollada por el Global Grid Forum. El objetivo de esta especificación es facilitar la integración de las aplicaciones con los gestores DRM's.

Palabras Clave

GridWay, DRMAA, Grid, API, Olvidar, Enviar, Globus, DRM, Gestor de Colas, Aplicaciones sobre Grid.

Project Briefing

The project objective is the implementation of the DRMAA specification over the GridWay tool.

GridWay was developed in order to help grid technologies users. It enables software developers to forget tasks like scheduling, resource discovering, resource selection …

Grid*W*ay is an user-oriented workload management tool (meta-scheduler) that allows an unattended and efficient execution of jobs on Computational Grids in *a submit and forget* fashion. Grid*W*ay is intended for being used with the Globus Toolkit components.
DRMAA (Distributed Resource Management Application API) is an API specification for the submission and control of jobs to one or more Distributed Resource Management (DRM) systems, developed by the Global Grid Forum. The scope of this specification is all the high level functionality which is necessary for an application to consign a job to a DRM system, including common operations on jobs like termination or suspension.

KeyWords

GridWay, DRMAA, Grid, API, Forget, Submit, Globus, DRM, Workload Manager , Grid based applications .

Introducción a GridWay

Qué es Grid

El término Grid Computing apareció a mediados de los años 90 en los trabajos de Ian Foster y Carl Kesselman. La idea de Grid se inició enfocada fundamentalmente al acceso remoto a recursos computacionales, buscando ser un paradigma de desarrollo sin centrarse en una tecnología concreta. Por su sencillez y concreción nos puede ayudar mencionar la definición de Ian Foster, co-director de la GriPhyN (*GridPhysics Networks*), que nos define o reconoce una Grid, como un sistema que coordina recursos no sujetos a control centralizado, usa protocolos e interfaces estándar, abiertos y de propósito general, y entregan una calidad de servicio importante.

Hoy en día, muchos centros de investigación y empresas tienen la necesidad de trabajar con programas de una gran complejidad, que requieren un gran número de ciclos de procesamiento o el acceso a mucha cantidad de datos, y que necesitan máquinas muy potentes para su ejecución (por ejemplo en campos como la astrofísica, la física molecular, centrales nucleares, ingenierías, grandes centros de desarrollo...). Esta necesidad, obligaría en un principio a muchas empresas a disponer de grandes servidores y computadores extremadamente caros, o en su lugar, máquinas más sencillas que necesitarían un tiempo excesivo para realizar los cálculos. De aquí nace la idea de computación bajo una Grid. El concepto de Grid es que varios centros de investigación comparten sus recursos en red, para crear un único sistema integrado, lo cual proporciona un aumento en el número de recursos de almacenamiento y procesamiento disponibles para los usuarios del Grid. Esta nueva tecnología es análoga a las redes de suministro eléctrico: la idea es ofrecer un único punto de acceso a un conjunto de recursos distribuidos geográficamente (supercomputadores, clusters, almacenamiento, fuentes de información, instrumentos, personal...). De este modo, los sistemas distribuidos se pueden emplear como un único sistema virtual en aplicaciones intensivas en datos o con gran demanda computacional. La evolución de las redes de comunicaciones de alta velocidad dedicadas a la investigación han creando un escenario idóneo para el despliegue de esta tecnología.

La idea de Grid surge por tanto, de la necesidad por la creciente demanda computacional actual, y también como una buena forma de aprovechar el gran número de redes globales existentes.

Permite interconectar recursos en diferentes dominios de administración respetando sus políticas internas de seguridad y su software de gestión de recursos en la Intranet.

Grid se diferencia de los sistemas cliente-servidor y otras tecnologías actuales (CORBA, EJB o .NET), en que está orientada a los recursos computacionales y no a la información, la seguridad no está en un segundo plano y la comunicación es asíncrona.

Se trata de un paso más allá de Internet, ya que incorpora un gran ancho de banda, alta velocidad de procesado, y bases de datos de gran tamaño. La interconexión entre ordenadores usando la tecnología Grid permite a un número elevado de usuarios obtener capacidad de procesamiento sin determinar a qué ordenador quieren conectarse. Por tanto, Grid se podría equiparar a capacidad de procesamiento de datos como Internet a capacidad de obtención de información.

Objetivo

El objetivo de la tecnología Grid, y el contexto en el que surge la idea, es el de *simular* un gran servidor o multicomputador al que se pueda tener acceso desde diferentes ubicaciones. De este modo lo que se pretende es proporcionar un alto rendimiento desde cualquiera de los centros unidos al Grid. El Grid proporciona una forma de tener acceso a esos recursos necesarios sin necesidad de los elevados costes que supondría una maquina muy potente.

Respecto al objetivo final de esta tecnología, queda todavía un largo camino que recorrer. Aunque actualmente ya existen pequeñas redes Grid, el objetivo final sería formar un Grid Mundial en el que todos los ordenadores del mundo estuvieran conectados de este modo, y así todos los usuarios tendrían acceso a la capacidad de cómputo y de almacenamiento que precisen sin preocuparse de donde se genera.

Utilidades

La tecnología Grid, tiene múltiples utilidades, tanto desde el punto de vista científico como el empresarial. Las infraestructuras Grid permiten cumplir con las demandas de computación y gestión de datos de las grandes aplicaciones y además facilitan la compartición de recursos para resolver demandas puntuales. Del mismo modo ha dado lugar a numerosas ideas de negocio, como el desarrollo de organizaciones virtuales, la comercialización en demanda de recursos, etc..

En definitiva, se trata de una tecnología que muchos analistas han definido como la siguiente revolución de Internet.

Aportaciones de Grid Computing

Las principales aportaciones de la Grid Computing son:

- Aumento en el rendimiento, y por tanto, aumento en la productividad de la empresa.

- Aprovechamiento de los recursos existentes, muchas veces desocupados, que existan en otras empresas.

- Ahorro considerable en gastos, ya que un equipo muy potente sólo se necesitará en ocasiones puntuales, o para aplicaciones concretas, estando "de más" el resto del tiempo.

- Desarrollo de aplicaciones desde diversos puntos geográficos, dando la oportunidad así, a las empresas, de estar ubicadas donde quieran pudiendo compartir recursos, información y trabajo. Esto provocará una mayor velocidad en el desarrollo de aplicaciones y de investigaciones.

Futuro de Grid

Aunque la computación Grid ha tenido un fuerte auge en los últimos años, esta evolución todavía no se ha plasmado en un uso real y amplio por parte de la comunidad científica y otros sectores.

Sin embargo el concepto de Grid Computing ha desbordado el ámbito académico y ha aparecido en multitud de medios de comunicación, asociando Grid Computing con términos como Internet, explotación de ciclos de CPU inutilizados, bases de datos e Inteligencia Artificial. Además muchas empresas han contribuido a su expansión proponiendo trabajos cuya base es el Grid Computing

Hoy en día, la computación Grid está en plena fase de desarrollo, y tenemos el pleno convencimiento que será la base de la computación del futuro. Hay numerosos proyectos en desarrollo, y grandes expectativas depositadas en ellos. Pero el desarrollo y utilización de recursos en Grid, también lleva implícito el uso de nuevas tecnologías y el desarrollo de aplicaciones que consigan integrar todos los computadores disponibles. Por ejemplo, GridWay.

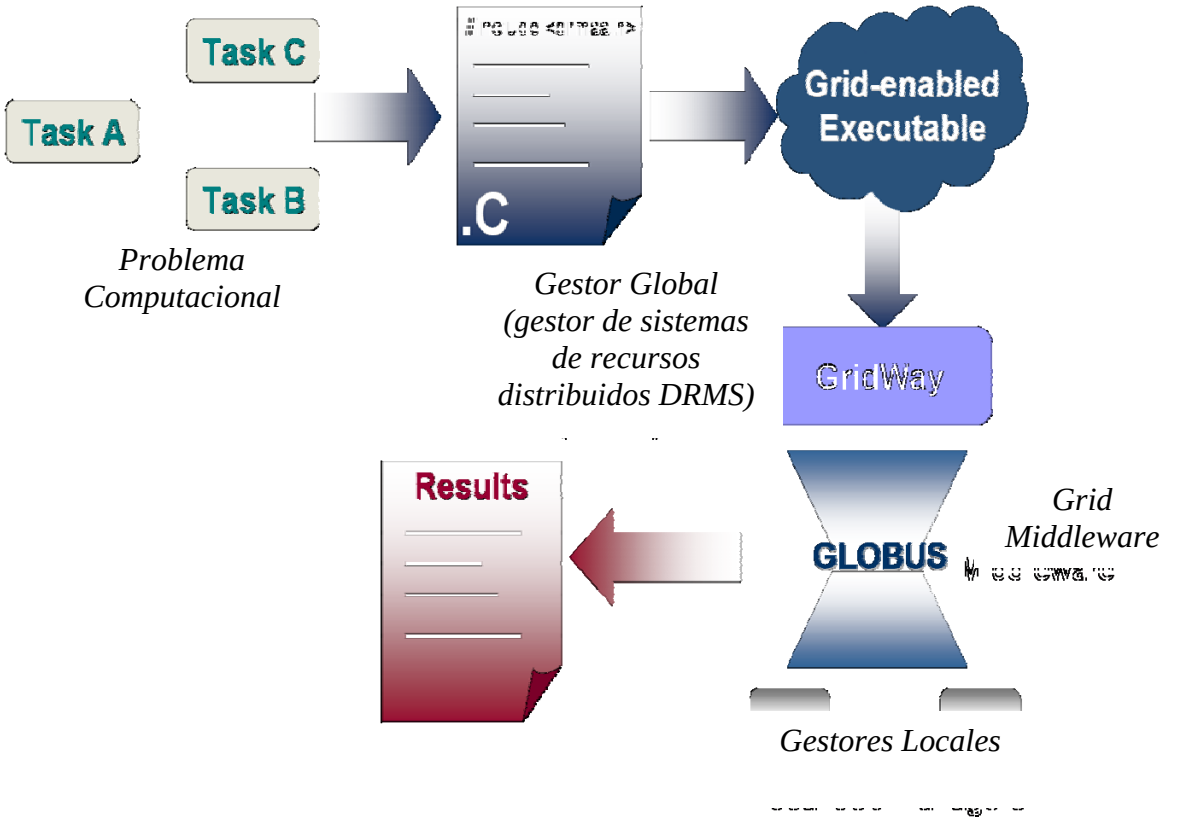
Qué es GridWay

GridWay es un framework de envío a Globus que permite un uso y ejecución más simple de trabajos en entornos Grid dinámicos.

GridWay es un sistema DRMS *(Distributed Resource Manager Systems). ¿Qué quiere decir esto*? Pues bien, para que lo entendamos, GridWay realiza automáticamente todos los pasos de planificación de los jobs, monitorización de trabajos, descubrimiento de recursos óptimos y adaptación de la ejecución y planificación de éstos a las condiciones del Grid en cada momento, proporcionando un rendimiento óptimo. Un job enviado puede ser reubicado si el sistema remoto falla o tiene un bajo rendimiento.

GridWay trabaja por encima de Globus, conectando redes heterogéneas con distintas estructuras y diversos programas de gestión (PBS, SunGridEngine....), proporcionando una visión transparente de todo esto al usuario final, que sólo tendrá que preocuparse de lanzar el job.

El recorrido de un job enviado a GridWay se puede entender mejor con el siguiente esquema:



Arquitectura GridWay

Una explicación mas detallada de las partes que componen GridWay con cada una de sus funcionalidades se explica a continuación:

- Agente de envió de jobs. Formado por:
    - Gestor de recursos (*Request Manager (RM)*): Modulo encargado de manejar las peticiones de los diferentes clientes de GridWay.
    - Gestor de expedición *(Dispatch Manager (DM))*: Módulo encargado de ejecutar la planificación de los jobs.
    - Gestor de Envío *(Submission Manager (SM))*: Módulo encargado de la ejecución y migración de los jobs.
    - Monitor de Rendimiento *(Performance Monitor (PM))*: Módulo encargado de evaluar el rendimiento de los trabajos.

- Selector de Recursos *(Resource Selector (RS))*: Usado por el dispatch manager para seleccionar el host más adecuado para ejecutar el job. Para realizar esta elección se tiene en cuenta la arquitectura, la carga de trabajos de cada máquina y diferentes parámetros.

- Drivers de acceso al Middleware (*Middleware Access Driver (MAD)):* Usado por el submission manager para proporcionar una interfaz con el gestor de recursos del host.

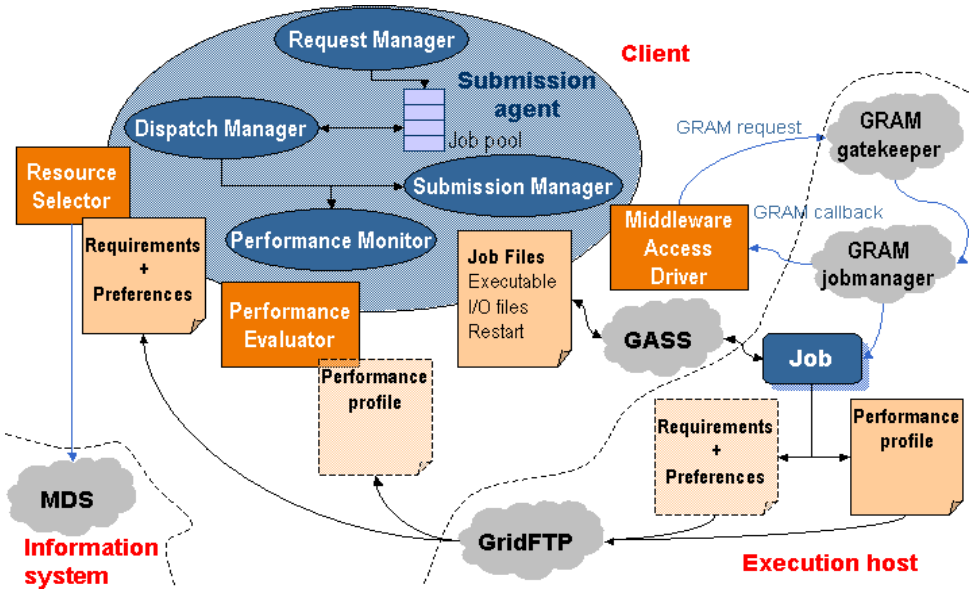
- Evaluador de rendimiento (*Performance Evaluator (PE)*): Usado por el monitor de rendimiento para verificar el progreso del job.

- Prolog: Usado por el submission manager para crear los directorios necesarios en el host remoto, preparar el ejecutable y transferir los archivos de entrada y los ejecutables a la maquina seleccionada.

- Wrapper: Usado por el submission manager para ejecutar el job y obtener su código de salida.

- Epilog: Usado por el submission manager para transferir los archivos de salida al host inicial, y eliminar de la máquina remota todos los directorios creados y liberar los recursos utilizados para la ejecución recién finalizada.

Un esquema que resume a grandes rasgos la arquitectura de GridWay, es el que se presenta a continuación:



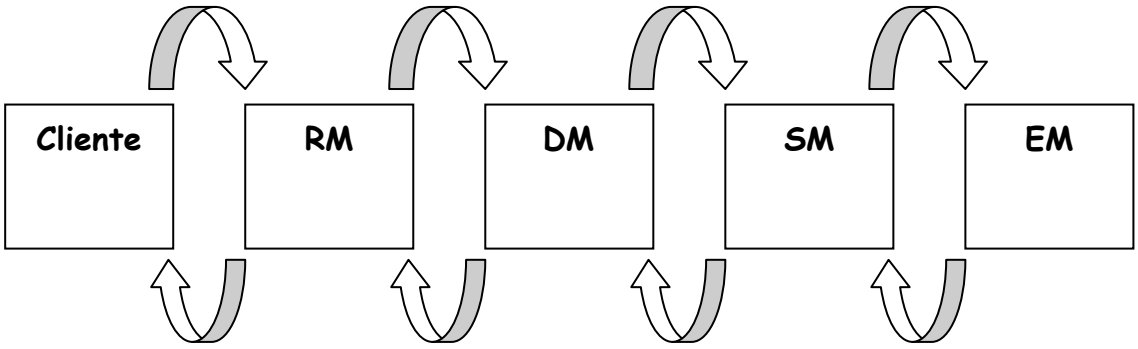
Comunicación entre módulos

Como hemos explicado en el punto anterior, GridWay tiene una organización modular, en la que cada módulo se encarga de una parte bien definida del proceso total de envío y gestión de las tareas. Pero cada uno de estos módulos tiene que comunicarse con el resto, ya que un trabajo va pasando por cada uno de los módulos desde que un cliente realiza una petición de ejecución de una orden hasta que dicha orden es ejecutada y finalizada correctamente.

Para ello, GridWay se basa en dos mecanismos de comunicación:

- Mensajes IPC
  Es el mecanismo utilizado para la comunicación entre el cliente y el Request Manager. Éste último tiene una cola en la que los clientes van enviando sus peticiones, y el Request Manager las va sacando de ahí para atenderlas. Las peticiones se van procesando en orden.

- Uso de señales:
  Cada uno de los módulos de GridWay dispone de un manejador de señales y una serie de señales registradas, que son las que usan cada uno de los módulos para comunicarse entre sí. Esta comunicación, sigue un orden predefinido, es decir, no cualquier modulo se comunica con cualquier módulo, sino que toda orden enviada al sistema sigue esta cadena:



La comunicación funciona de la siguiente manera: Al inicializarse cada uno de los módulos, se registra todas las acciones deseadas junto con su manejador (función que se ejecutará cuando se reciba una señal de este tipo. A continuación se arranca un *"listener"*, método que esta continuamente "escuchando", y que cuando detecta que se le ha enviado una señal, lanza el manejador que dicha señal tenga asociado.

Cuando un módulo quiere comunicarse con otro, por ejemplo el Request Manager con el Dispatch Manager, simplemente lanzará la señal apropiada.

La organización de la comunicación que tiene GridWay, hace que no sea posible "perdidas " de mensajes, ya que el camino que tiene que atravesar es de ida y vuelta. Es decir, una petición va desde el Cliente al EM, pero cada uno de los módulos una vez que ha atendido la señal que les envió el módulo anterior, deberá lanzar él una señal de confirmación al primer módulo, para informarle que la acción deseada ha sido realizada correctamente, o bien, que se ha llegado al estado que fuera. Así, este primer módulo sabrá que acción debe realizar en función de lo que el módulo al que mandó la señal haya conseguido. Así, por ejemplo, un job no se eliminará ni se liberará la memoria que tiene asignado aunque haya finalizado la ejecución hasta que no le llegue al dispatch manager la señal que se lo indique, y a su vez, el Dispatch informará de ello al Request Manager, que podrá atender una nueva petición de la cola.

A continuación, enumeramos las acciones que cada uno de los módulos tiene registradas:

- Request Manager:
  - GW_ACTION_FINALIZE con el método asociado gw_rm_finalize()
  - GW_RM_MSG con el manejador asociado gw_rm_msg()
  - GW_RM_CALLBACK con el método asociado gw_rm_callback()
- Dispatch Manager:
  - GW_ACTION_FINALIZE con el método asociado gw_dm_finalize()
  - GW_ACTION_TIMER con el manejador asociado gw_dm_schedule()
  - GW_DM_SUBMIT con gw_dm_submit()
  - GW_DM_RELEASE con el método gw_dm_release()
  - GW_DM_HOLD con el método gw_dm_hold()
  - GW_DM_SUBMIT_ARRAY con el método gw_dm_submit_array()

  - GW_DM_WAIT con el método gw_dm_wait()
  - GW_DM_KILL con el método gw_dm_kill()
  - GW_DM_HARD_KILL con el método gw_dm_hard_kill()
  - GW_DM_STOP con el método gw_dm_stop()
  - GW_DM_RESUME con el método gw_dm_resume()
  - GW_DM_RESCHEDULE con el método gw_dm_reschedule()
  - GW_DM_DONE con el método gw_dm_done()

- Submission Manager:

  - GW_ACTION_FINALIZE con el manejador gw_sm_finalize()
  - GW_SM_SUBMIT con el manejador gw_sm_submit()
  - GW_SM_FAIL con el manejador gw_sm_fail()
  - GW_SM_STOP con el manejador gw_sm_stop()
  - GW_SM_KILL con el manejador gw_sm_kill()
  - GW_SM_HARD_KILL con el manejador gw_sm_hard_kill()
  - GW_SM_DONE con el manejador gw_sm_done()
  - GW_SM_MIGRATE con el manejador gw_sm_migrate()
  - GW_SM_WRAPPER con el manejador gw_sm_wrapper()
  - GW_SM_EPILOG con el manejador gw_sm_epilog()
  - GW_SM_MIGRATION_PROLOG con el manejador gw_sm_migration_prolog()
  - GW_SM_MIGRATION_EPILOG con el manejador gw_sm_migration_epilog()

- Execution Manager:
  - GW_ACTION_FINALIZE con el manejador gw_em_finalize()
  - GW_ACTION_TIMER con el manejador gw_em_poll()
  - GW_EM_SUBMIT con el manejador gw_em_submit()
  - GW_EM_CANCEL con el manejador gw_em_cancel()
  - GW_EM_STATE_PENDING con el manejador gw_em_pending()
  - GW_EM_STATE_ACTIVE con el manejador gw_em_active()
  - GW_EM_STATE_SUSPENDED con el manejador gw_em_suspended()
  - GW_EM_STATE_DONE con el manejador gw_em_done()
  - GW_EM_STATE_FAILED con el manejador gw_em_failed()

DRMAA

Qué es DRMAA

Distributed Resource Management Application API (DRMAA) es una especificación desarrollada por el Global Grid Forum para facilitar la integración de los Gestores de Recursos Distribuidos (DRMS) con aplicaciones, permitiendo a los programadores, mediante una librería de instrucciones estándar, la gestión, envío y control de trabajos, a los DRMS sin la necesidad de un conocimiento explicito de las características propias de éstos. Ayudando por tanto, no solo al desarrollo de estas aplicaciones a mejorar la portabilidad de estos códigos entre diferentes DRMS de forma transparente al programador.

DRMAA especifica mecanismos para enviar trabajos, monitorizar y controlar dichos trabajos y obtener su estado final.

Respecto a los beneficios que aporta DRMAA podemos destacar un rápido desarrollo de las aplicaciones distribuidas, la oportunidad para nuevas aplicaciones, mejoras en los sistemas de gestión de recursos y portabilidad de las aplicaciones distribuidas.

La especificación está definida para poder ser implementada en múltiples lenguajes. La especificación de hecho, esta definida usando un lenguaje IDL

Consideraciones de Implementación

El documento de especificación indica una serie de características que cualquier implementación del estándar sería deseable que cumplieran:

Portabilidad

La implementación de DRAMA debería estar desarrollada en módulos que pueden ser intercambiables en tiempo de ejecución por el usuario. En último caso, la aplicación DRMAA podría ejecutarse en un sistema DRMS específico mediante la configuración de un conjunto de variables de entorno para ese sistema DRMS, o proporcionando una cadena de conexión específica para el sistema DRMS en el paso de inicialización.

Thread Safety

Los autores esperan que los desarrolladores de la librería DRMAA proporcionen multithread. Por ello es recomendable que la librería DRMAA sea thread-safety, y permita aplicaciones multihilo que usen la librería DRMAA sin una sincronización explicita entre los hilos de la aplicación.

Se recomienda que los implementadores de DRMAA califiquen sus implementaciones como thread-safe de acuerdo con el criterio anterior. Los desarrolladores de implementaciones que no sean thread-safe deberían documentar todas las interfaces "inseguras" y proporcionar una lista de interfaces y sus dependencias con las rutinas externas "inseguras". Sin embargo, antes de que una aplicación multihilo pueda usar cualquier interfaz DRMAA la rutina de inicialización de la librería debe ser llamada por un único hilo, normalmente el principal, y del mismo modo, la desconexión de la librería será realizada por un único hilo.

Sincronización

DRMAA gestiona el asincronismo del envío y terminación de trabajos de forma similar a los procesos Unix, bloqueando mediante una llamada wait el job deseado.

Entorno de la aplicación distribuida

DRMAA especifica mecanismos para enviar, monitorizar y controlar un job, y obtener su estado final. Idealmente, las implementaciones DRMAA y las aplicaciones distribuidas no necesariamente están configuradas para un particular entorno DRMS o una política DRMS específica. Para facilitar el desarrollo donde éste no este completamente acoplado, las categorías de jobs y la especificación nativa debe ser usada para abstraer o agregar las políticas específicas con simples cadenas que serán interpretadas por las implementaciones DRMAA.

Descripción General de las Rutinas

Las rutinas del API están divididas en cinco categorías:

- INIT y EXIT
- Gestión de los *job templates*
- Envío de trabajos
- Monitorización y Control de Trabajos
- Funciones Auxiliares

Init y Exit

Esta categoría esta compuesta únicamente por dos rutinas, init y exit.

int drmaa_init(string contact, drmaa_context_error_buf)

La rutina init es la encargada de establecer e inicializar la comunicación con los diferentes DRMS, inicializando la librería API DRMAA y creando una nueva sesión DRMAA (DRMAA session). La especificación de DRMAA recomienda que no se permita el uso de sesiones anidadas sino que se libere la sesión antes de inicializar otra.

Para que el resto de rutinas, salvo ciertas funciones auxiliares - como drmaa_version(), drmaa_get_DRM_system(), drmaa_get_DRMAA_implementation(), drmaa__strerror(), o drmaa_get_contact() - funcionen, debe haberse iniciado previamente la sesión, es decir, que la llamada a la función INIT debe hacerse antes que la llamada a dichas funciones.

contact: Parámetro de entrada que indica qué sistema DRM usar
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_exit(drmaa_context_error_buf)

La rutina exit es la encargada de realizar la desconexión con los DRMS así como liberar los datos de la sesión DRMAA actual. Esta rutina, al igual que la de INIT, requiere parámetros que permitan una conveniente desconexión.
La llamada a esta función provoca el término de la sesión DRMA, pero no afecta al estado de los trabajos.

drmaa_context_error_buf : Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

En el caso que la implementación esté destinada al uso de *threads* DRMAA recomienda que, además que la implementación sea *thread-safe;* evitando a los programadores la sincronización explicita de los hilos, las rutinas init y exit deban ser ejecutadas por un único hilo, particularmente el *master thread*.

Gestión de Templates

Los trabajos remotos y sus atributos deben ser especificados por el parámetro del manejador del *job template* Los atributos del *job* deben ser un *string* o un vector de valores de tipo *string.*

El conjunto de atributos del *job* es el siguiente:

- Remote command to execute
- Remote command input parameters, a vector parameter
- Job state at submission
- Job environment, a vector parameter
- Job working directory
- Job category
- Native specification
- Standard input, output, and error streams
- E-mail distribution list to report the job completion and status, a vector parameter
- E-mail suppression
- Job start time
- Job name to be used for the job submission

La gestión de los *templates* se realiza en dos ámbitos:

- Asignación y borrado
- Funciones accesoras y modificadoras

Respecto a la **asignación y borrado**, se dispone de dos rutinas**:** drmaa_allocate_job_template y void drmaa_delete_job_template (job_template jt). La primera se encarga de la asignación de un nuevo *job template* y la segunda, de la acción contraria. Esta última rutina no tiene efecto sobre el estado de los trabajos.

job_template drmaa_allocate_job_template(job_template jt, drmaa_context_error_buf)

jt: Parámetro de entrada que se refiere al *job template*
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

void drmaa_delete_job_template(job_template jt, drmaa_context_error_buf)

jt: Parámetro de entrada que se refiere al *job template*
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

Respecto a las funciones accesoras y modificadoras se dispone de las siguientes:

int drmaa_set_attribute(job_template jt, string name, string value, drama_context_error_buf)

Función que añade el par (name, value) a la lista de atributos en el *job template* jt

jt: Parámetro de entrada que se refiere al *job template*
name: Parámetro de entrada que se refiere al nombre del atributo
value: Parámetro de entrada que se refiere al valor del atributo
drama_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_set_vector_attribute(job_template jt, string name, string array values, drama_context_error_buf)

Función que añade el par (name, values) a la lista de vectores de atributos en el *job template* jt

jt: Parámetro de entrada que se refiere al *job template*
name: Parámetro de entrada que se refiere al nombre del atributo
values: Parámetro de entrada consistente en un vector de valores de atributo
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

string drmaa_get_attribute(job_template jt, string name, string value, drmaa_context_error_buf)

Función que, si *name* es un atributo (no vector) existente en el *job template* jt, devuelve el valor *value* de *name;* en caso contrario devuelve NULL

jt: Parámetro de entrada que se refiere al *job template*
name: Parámetro de entrada que se refiere al nombre del atributo
value: Parámetro de salida que devuelve el valor del atributo *name* o NULL si éste no existe en el *job template.*
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

string array drmaa_get_vector_attribute(job_template jt, string name, string array values, drmaa_context_error_buf)

Función que, si *name* es un atributo (no vector) existente en el *job template* jt, devuelve el valor *value* de *name;* en caso contrario devuelve NULL

jt: Parámetro de entrada que se refiere al *job template*
name: Parámetro de entrada que se refiere al nombre del atributo
values: Parámetro de salida que devuelve los valores del atributo *name* o NULL si éste no existe en el *job template.*
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

drmaa_get_attribute_names(names, drmaa_context_error_buf)

Función que devuelve el conjunto de nombres de atributos soportados cuyo tipo es string, en el vector de tipo string *names.*

names: Parámetro de salida donde se escribe el nombre de los atributos (de tipo string).
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

drmaa_get_vector_attribute_names(names,drmaa_context_error_buf)

Función que devuelve el conjunto de nombres de atributos soportados cuyo tipo es vector string, en el vector de tipo string *names.*

names: Parámetro de salida donde se escribe el nombre de los atributos (de tipo vector string).
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

Envío de trabajos

Los trabajos enviados al sistema DRM están identificados mediante un identificador de *job* que, por razones de flexibilidad es de tipo *string*
Son dos las rutinas que están descritas en este ámbito, una para el envío de trabajos individuales y otra para el envío de un conjunto de trabajos.

int drmaa_run_job(string job_id, job_template jt, drmaa_context_error_buf)

Función que envía un trabajo con los atributos que están previamente definidos en el *job template* jt.

job_id: Parámetro de entrada que hace referencia al identificador del *job*
jt: Parámetro de entrada que se refiere al *job template*
drmaa_context_error_buf : Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_run_bulk_job(string array job_ids, job_template jt, int start, int end, int incr, drmaa_context_error_buf)

Función que envía un conjunto de trabajos paramétricos, cada uno con atributos definidos en el *job template* jt.

job_ids: Parámetro de entrada consistente en un vector de identificadores de *jobs*
jt: Parámetro de entrada que se refiere al *job template*
start: Parámetro de entrada que se refiere al índice del comienzo
end: Parámetro de entrada que se refiere al índice del final
incr: Cantidad añadida en cada ciclo de envío de trabajos. Por norma general suele ser 1 de forma que el número total de jobs del array son start-end.
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

Monitorización y Control de Trabajos

La monitorización y control de trabajos maneja varias funciones:

- *holding, releasing, suspending, resuming, y killing* de trabajos
- Control del código de salida del *job* finalizado
- Control del estado del *job* remoto
- Espera hasta el fin de la ejecución del *job* remoto.
- Espera hasta el fin de la ejecución de todos los *jobs* o de un subconjunto de los *jobs* de la sesión actual

Las señales de Unix y de Windows se remplazan con estas rutinas de control, que tienen su imagen en DRMS.

El *job* remoto estará en uno de los siguientes estados:

- *System hold*
- *User hold*
- *System and user hold simultaneously*
- *Queued active*
- *System suspended*
- *User suspended*
- *System and user suspended simultaneously*
- *Running*
- *Finished (un)successfully*

A un trabajo rechazado no se le asigna un identificador y, en consecuencia, no estará en ningún estado. Cabe destacar que en un sistema distribuido puede que no sea siempre posible determinar el estado de los trabajos remotos.

Las funciones disponibles son las siguientes:

int drmaa_control(string job_id, int action, drmaa_context_error_buf)

Función que comienza, detiene, reanuda, o mata el trabajo cuyo identificador es *job_id*.
Si *job_id es* DRMAA_JOB_IDS_SESSION_ALL entonces esta rutina actúa sobre todos los trabajos enviados durante esta sesión.

Para evitar carreras en aplicaciones multihilo el usuario de la implementación DRMAA debe sincronizar explícitamente esta llamada con cualquier otra llamada a envío de job o llamado de control que cambie el numero de jobs remotos.

job_id: Parámetro de entrada que se refiere al identificador del *job*
action: Parámetro de entrada que hace referencia a la acción de control
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

Esta función esperará hasta que la acción haya sido reconocida por el sistema DRM, pero no esperará necesariamente hasta que la acción se haya completado.

int drmaa_synchronize(string array job_ids, signed long timeout, boolean dispose,drmaa_context_error_buf)

Función que espera a que todos los trabajos especificados por los identificadores *job_ids* finalicen. Si *job_id es* DRMAA_JOB_IDS_SESSION_ALL entonces esta rutina actúa sobre todos los trabajos enviados durante esta sesión, es decir, espera hasta que todos ellos acaben.

ids: Parámetro de entrada que se refiere al identificador del *job*
timeout: Parámetro de entrada que indica el tiempo que se espera en la llamada a esta función, es decir, el tiempo de bloqueo. El tiempo del sistema deberá ser comprobado antes y después de la llamada a esta función, para poder comprobar cuánto tiempo ha pasado.
dispose: Parámetro de entrada. Si este parámetro es *false* la información del *job* permanece disponible y puede ser recuperada mediante la llamada a drmaa_wait(). Si el parámetro es *true*, la información no se mantiene.
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito. Si la invocación a la llamada finaliza después de consumido el tiempo *timeout*, el código devuelto será DRMAA_ERRNO_EXIT_TIMEOUT.

Para evitar carreras en aplicaciones multihilo el usuario de la implementacion DRMAA debe sincronizar explícitamente esta llamada con cualquier otra llamada a envío de job o llamado de control que cambie el numero de jobs remotos.
Para evitar el bloqueo indefinido en la llamada a esta función se utiliza el tiempo *timeout.* Un valor DRMAA_TIMEOUT_WAIT_FOREVER (-1) para este parámetro hará que se espere indefinidamente, es decir, hasta que el resultado esté disponible. El valor DRMAA_TIMEOUT_NO_WAIT (0) hará que se "vuelva" inmediatamente si el resultado no está disponible.

int drmaa_job_ps(string job_id, int remote_ps, drama_context_error_buf)

Función que devuelve el estado del trabajo cuyo identificador es *job_id*

job_id: Parámetro de entrada que hace referencia al identificador del *job*
remote_ps: Parámetro de salida que se refiere al estado del trabajo
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

Los valores posibles que se pueden devolver en el parámetro *remote_ps* son los siguientes:

DRMAA_PS_UNDETERMINED = 00H : no se puede determinar el estado del proceso
DRMAA_PS_QUEUED_ACTIVE = 10H : el *job* está en cola y activo
DRMAA_PS_SYSTEM_ON_HOLD = 11H : el *job* está en cola y en estado *system hold*
DRMAA_PS_USER_ON_HOLD = 12H : el *job* está en cola y en estado *user hold*
DRMAA_PS_USER_SYSTEM_ON_HOLD = 13H : el *job* está en cola y en los estados *system hold* y *user hold*
DRMAA_PS_RUNNING = 20H : el *job* está en ejecución
DRMAA_PS_SYSTEM_SUSPENDED = 21H : el *job* está en estado *suspend*
DRMAA_PS_USER_SUSPENDED = 22H : el *job* está en estado *user suspend*
DRMAA_PS_DONE = 30H : el *job* finalizó de manera correcta
DRMAA_PS_FAILED = 40H : el *job* finalizó, pero falló.

Dentro de este grupo de funciones se encuentran también las siguientes:

int drmaa_wait(job_id, int stat, signed long timeout, string array rusage, drmaa_context_error_buf)

Esta función espera a que el trabajo con el identificador *job_id* falle o finalice su ejecución. Si *job_id es* DRMAA_JOB_IDS_SESSION_ALL entonces esta rutina actúa sobre todos los trabajos enviados durante esta sesión, es decir, espera hasta que todos ellos acaben.

job_id: Parámetro de entrada que se refiere al identificador del *job*
este parámetro devuelve a su vez el identificador del trabajo finalizado. Puede ser NULL.
stat: Parámetro de salida que se refiere al código del estado del *job*
timeout: Parámetro de entrada que se refiere al tiempo de bloqueo de la función. El tiempo del sistema deberá ser comprobado antes y después de la llamada a esta función, para poder comprobar cuánto tiempo ha pasado.
rusage: nombre del recurso
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito. Si la invocación a la llamada finaliza después de consumido el tiempo *timeout*, el código devuelto será DRMAA_ERRNO_EXIT_TIMEOUT.

Al igual que en la función *drmaa_synchronize,* para evitar el bloqueo indefinido en la llamada a esta función se utiliza el tiempo *timeout.* Un valor DRMAA_TIMEOUT_WAIT_FOREVER (-1) para este parámetro hará que se espere

indefinidamente, es decir, hasta que el resultado esté disponible. El valor DRMAA_TIMEOUT_NO_WAIT (0) hará que se "vuelva" inmediatamente si el resultado no está disponible.

int drmaa_wifexited(OUT exited, IN stat, OUT drmaa_context_error_buf)

Función que devuelve en el parámetro *exited* si el código *stat* corresponde a un trabajo que terminó de manera normal.

exited: Parámetro de salida que devuelve un valor distinto de 0 si el código de *stat* corresponde al código de salida de un trabajo que terminó de manera normal. Si el valor es 0 quiere decir que puede que el trabajo terminara normalmente, pero que el código de estado no está disponible o que no se sabe si el trabajo terminó bien o no. En ambos casos esta función no proporciona información sobre el estado de salida.
stat: Parámetro de entrada con el código de salida de un *job*.
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_wexitstatus(OUT exit_status, IN stat, OUT drmaa_context_error_buf)

Función que evalúa en el parámetro *exit_status* el código de salida almacenado en *stat* después de una llamada a la función drmaa_wifexited.

exit_status: Parámetro de salida en el que escribe el código de salida del *job*
stat: Parámetro de entrada con el código de un trabajo finalizado
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_wifsignaled(OUT signaled, IN stat, OUT drmaa_context_error_buf)

Función que evalúa en el parámetro *signaled* si el trabajo terminó debido a la recepción de una señal.

signaled: Parámetro de salida. Si devuelve un valor distinto de cero es porque el trabajo terminó debido a la recepción de una señal. Si devuelve 0 puede indicar que el trabajo terminó a causa de una señal pero ésta no está disponible, o bien que no se sabe si el trabajo terminó por esta causa. En ambos casos esta función no proporcionará información sobre la señal.
stat: Parámetro de entrada que contiene el código de salida del *job.*
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_wtermsig(OUT signal, IN stat, OUT drmaa_context_error_buf)

Función que evalúa en el parámetro *signal*, después de la llamada a la función wifsignaled(stat), la representación de la señal que causó la terminación de un trabajo.

signal: Parámetro de salida en el que escribe la representación de la señal.
stat: Parámetro de entrada con el código de un trabajo finalizado
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_wcoredump(OUT core_dumped, IN stat, OUT drmaa_context_error_buf)

Función que evalúa en el parámetro *core_dumped*, si el valor devuelto por una llamada a la función drmaa_wifsignaled(stat) no es cero, si se creó una imagen *core* del trabajo finalizado.

core_dumped: Parámetro de salida que indica si se creó la imagen.
stat: Parámetro de entrada que se evalúa para saber si se creó una imagen o no.
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_wifaborted(OUT aborted, IN stat, OUT drmaa_context_error_buf)

Función que evalúa en *aborted*, si el código de *stat* se corresponde con un *job* que finalizó antes de entrar en su estado de ejecución.

aborted: Parámetro de salida que indica si el *job* finalizó antes de ejecutarse.
stat: Parámetro de entrada que se corresponde con el código de salida de un trabajo.
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

Funciones Auxiliares

const char error_string drmaa_strerror (int errno);

Función que devuelve el código de error asociado al número de error especificado en *errno*. Si éste es no válido, entonces devuelve NULL.

errno: Parámetro de entrada que contiene un código de error.
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_get_contact(char *contacts, drmaa_context_error_buf);

Si se llama antes de una llamada a la función drmaa_init, devolverá una cadena separada por comas formada por los *contactos* de implementaciones proporcionadas por DRM. Si por el contrario se llama después de la función drmaa_init, devuelve la cadena de contacto del el sistema para el que se inicializó la librería.

contacts: Parámetro de salida donde se devuelve la información.
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

int drmaa_version(int major, int minor, drmaa_context_error_buf)

Función que devuelve los números de versión *major* y *minor* de la librería DRMAA. Para DRMAA 1.0, *major* es '1' y *minor* es '0'.

major : Parámetro de salida donde se escribe el número de versión mayor
minor: Parámetro de salida donde se escribe el número de versión menor
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

drmaa_get_DRM_system(string drm_systems, drmaa_context_error_buf)

Si se llama a esta función antes de la llamada a la función drmaa_init, devuelve una cadena separada por comas con la lista de identificadores de sistema DRM. Si se llama después de la función drmaa_init, devuelve el sistema DRM seleccionado.

drm_systems: Parámetro de salida donde se escribirá el/los identificadores de sistema.
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

drmaa_get_DRMAA_implementation(string drmaa_implementations, drmaa_context_error_buf)

Si la llamada a esta función se realiza antes de la llamada a drmaa_init, devuelve una cadena delimitada con comas con la lista de implementaciones proporcionadas por DRMAA. En caso de que se llame a esta función después de una llamada a drmaa_init, devuelve la implementación seleccionada.

drmaa_implementation: Parámetro de salida donde se escribirá la información sobre la implementación.
drmaa_context_error_buf: Parámetro de salida que contiene el código de error o DRMAA_ERRNO_SUCCESS en caso de éxito.

Nuestro Proyecto

¿Qué pretendemos?

Este proyecto no es un proyecto creado desde cero por nosotros mismos, sino que consiste en la colaboración en el proyecto GridWay, desarrollado por el departamento ACYA, y que se está llevando a cabo gracias a la colaboración de la Universidad Complutense de Madrid (UCM), el Centro de Astrobiología(CAB) y el Instituto Nacional de Técnica Aeroespacial (INTA). Dicho proyecto, de gran envergadura, está en plena fase de desarrollo y, aunque a la finalización de nuestro proyecto particular esté aún inacabado, esperamos contribuir con nuestro trabajo a la construcción de lo que creemos será una potente herramienta de gestión de recursos distribuidos (DRMS).

El objetivo del proyecto es dotar a la herramienta GridWay de una implementacion completa del estándar de programación DRMAA, ya que con ello se dota a la aplicación de un mayor potencial y un aumento en la facilidad de trabajar con dicha aplicación a los usuarios.

Se pretende que el programador tenga una visión transparente de todo lo relacionado con la elección de recursos, gestión de trabajos, ... dedicándose simplemente a enviar el trabajo al sistema DRM y olvidarse. Ésta técnica se denomina "submit and forget" y gracias a ella GridWay realiza automáticamente todos los pasos de la planificación de los trabajos (selección de recursos, preparación, envío, monitorización, migración y finalización).

Cabe destacar que, por otro lado, que Grid carece de paradigmas de programación estándar. El grupo de trabajo DRMAA (DRMAA-WG) ha desarrollado una especificación API para el envío de trabajos, monitorización y control que proporciona una interfaz de alto nivel para DRMS.

¿Qué nos encontramos?

La versión inicial de GridWay que nos es entregada cuando comenzamos a trabajar, es la versión 4.0. Esta versión, es una versión funcional, ya que ya es posible enviar trabajos y se realiza una gestión de los recursos disponibles, pero incompleta según los objetivos que dicha herramienta tiene marcados, y que son necesarios para poder realizar una implementación completa del estándar.

Dicha versión ya tiene implementada una implementación inicial y también incompleta del estándar DRMAA, en la cual faltan por desarrollar muchas de las funciones que se indican en la especificación del Global Grid Forum, y muchas otras están implementadas pero con falta de funcionalidad.

Uno de los *"problemas"* que nos encontramos a la hora de ponernos a desarrollar las funciones que faltaban en DRMAA es que muchas de ellas eran imposibles de implementar con la versión inicial de GridWay que se nos había facilitado. Por ello, nuestro proyecto pasó de estar centrado única y exclusivamente en la implementación del estándar, a realizar dicha implementación dotando a GridWay de toda la funcionalidad necesaria para poder desarrollar el estándar. Esto conllevó la creación de nuevas señales, nuevos estados de los jobs..... que se explicará en los apartados sucesivos.

Nuestros primeros pasos, por tanto, fueron:

- Entender bien el DRMAA 1.0 y su implementación inicial en GridWay.
- Revisar la versión inicial comparándola con la especificación para saber qué es exactamente lo que hay que añadir o modificar.
- Entender la parte del código de GridWay implicada en nuestros objetivos.

Una vez completados estos pasos, se elaboró un informe en el que se detallaba:

- Funcionalidades incompletas de las funciones más destacadas
- Posibles errores encontrados
- Preguntas surgidas a partir de la revisión del código existente

Posteriormente, se hizo una reunión con el director del proyecto en la cual se nos indicó más claramente las posibles ampliaciones a realizar en la versión inicial.
Éstas son las más destacadas:

- Nuevos estados hold y release
- Implementación de la rutina gw_wait con *timeout*
- Uso de *thread-safe* para DRMAA
- Soporte para el atributo START_TIME
- Implementación de la señal HARDKILL

Estos puntos se explicarán con detalle en secciones posteriores. Cabe destacar que, si bien éste fue un primer boceto de los objetivos del proyecto, durante la realización del mismo se fue perfilando y se fueron añadiendo nuevas tareas.

¿Qué hemos conseguido?

Como indicamos en el comienzo, nuestro proyecto forma parte de un proyecto de grandes dimensiones, y por tanto, ni partimos de cero inicialmente, ni hemos concluido totalmente todo el trabajo que se podía hacer finalmente, pero sí la mayor parte de los objetivos que nos habíamos planteado en un principio. Hemos terminado la implementación del estándar DRMAA para GridWay, siguiendo las especificaciones marcadas por los autores del estándar, y además, hemos ampliado la implementación de GridWay, ya que en muchos casos ha sido necesario para completar alguna de las funciones DRMAA, ya que la aplicación(GridWay) estaba incompleta o carecía de la funcionalidad necesaria. Nuestro trabajo en GridWay ha estado centrado en todo lo relacionado con el envío y gestión de trabajos, principalmente en los módulos que van desde que el cliente realiza alguna acción hasta que el job se lanza a ejecución. A partir de ahí, no tocamos nada, ya que como dijimos GridWay es un proyecto de grandes dimensiones y esa zona cae fuera de nuestros objetivos. Por tanto, todos los cambios realizados por nuestro grupo de trabajo han sido en el modulo gw_client, donde están definidas las diferentes funciones que los usuarios pueden usar en la aplicación, en el modulo request manager, que es el encargado de atender las peticiones de los clientes, y en el modulo dispatch manager, que es el encargado de crear los jobs, lanzarlos a ejecución y en cierta manera controlarlos. Únicamente para la implementacion de la señal hard kill llegamos a tener que tocar código del submission manager, pero muy superficialmente.

La implementación completa del estándar DRMAA supone aportar una mayor comodidad a los programadores a la hora de trabajar con GridWay, ya que no se tienen que preocupar de conocer internamente el funcionamiento de GridWay, ni la interfaz mas básica de jobs del sistema DRMS, resultando por tanto toda la arquitectura interna de éste transparente para el usuario, y proporcionándole un modelo de programación mucho más fácil de usar.

Además, el uso del estándar facilitará en gran medida la integración de diferentes aplicaciones.

Desarrollo

Comienzo: Adquisición de Conocimiento

Como hemos contado en la introducción, los primeros pasos que tuvimos que llevar a cabo fueron:

- Entender el DRMAA version 1.0
- Revisar la implementación inicial en GridWay, comparándola con la especificación para saber qué es exactamente lo que hay que añadir o modificar.
- Entender la parte del código de GridWay implicada en nuestros objetivos.

Entender DRMAA v1.0

El estándar dispone de una página propia, http://www.drmaa.org, donde se puede encontrar tanto el documento de especificación (actualmente versión 1.0) como una especificación para C (C Binding[1]).

Lo primero que tuvimos que hacer para saber que nos traíamos entre manos, fue leer y entender bien estas especificaciones, para conseguir entender el modo de funcionamiento de DRMAA y lo que se pretende.

Se puede destacar de esta fase, que aprendimos cómo maneja los trabajos DRMAA (a través de la definición de job template), el manejo de errores, a través de constantes ya definidas que nos indican la causa del error en cada momento, los atributos necesarios de los que debe estar dotado un job, y el entendimiento de las funciones para enviar y gestionar trabajos.

Revisar la implementación inicial de DRMAA en GridWay

Una vez entendido el objetivo de del DRMAA, pasamos a revisar la implementación inicial que se nos entrega, viendo qué funciones hay que modificar o ampliar.
Posteriormente se elaboró un inventario con una lista de cambios posibles con el fin de aumentar la funcionalidad existente hasta el momento.

Un esquema de este inventario es el siguiente:

- Funciones ya implementadas con funcionalidad incompleta o bien susceptibles de ser modificadas:
    - drmaa_get_next_attr_value:

[1] Actualmente versión 1.0, aunque al comienzo del proyecto estaba disponible la versión 0.95

- Posible cambio del código de error devuelto: en lugar de DRMAA_ERRNO_INTERNAL_ERROR, DRMAA_NO_MORE_ELEMENTS.

- Funcionalidad completa

- drmaa_get_next_job_id:

  - Posible cambio del código de error devuelto: en lugar de DRMAA_ERRNO_INTERNAL_ERROR, DRMAA_NO_MORE_ELEMENTS.

  - Funcionalidad completa

- drmaa_init:

  - Posible cambio del código de error devuelto: en lugar de DRMAA_ERRNO_INVALID_CONTACT_STRING, DRMAA_ERRNO_INVALID_ARGUMENT, y en lugar de DRMAA_ERRNO_DRM_COMMUNICATION_FAILURE, DRMAA_ERRNO_DRMS_INIT_FAILED.

  - Funcionalidad: El parámetro *contact* se utiliza para indicar a qué sistema DRM se conecta. En la implementación actual, el único valor posible para *contact* es NULL.

- drmaa_exit:

  - Posible cambio del código de error devuelto: Esta función debe devolver error cuando no hay ninguna sesión activa (DRMAA_ERRNO_NO_ACTIVE_SESSION). Sin embargo, en esta implementación devuelve siempre DRMAA_SUCCESS.

  - Funcionalidad completa

- drmaa_delete_job_template:

  - Posible cambio del código de error devuelto: En lugar de DRMAA_ERRNO_INVALID_JOB, DRMAA_ERRNO_INVALID_ARGUMENT, porque el primero no aparece en la especificación como código de retorno.

  - Funcionalidad completa

- drmaa_set_attribute:

  - Posible cambio del código de error devuelto: En lugar de DRMAA_ERRNO_INVALID_JOB,

DRMAA_ERRNO_INVALID_ARGUMENT, por la misma razón que en el caso anterior.

- drmaa_get_attribute:
  - Posible cambio del código de error devuelto: En lugar de DRMAA_ERRNO_INVALID_JOB, DRMAA_ERRNO_INVALID_ARGUMENT, por la misma razón que en el caso anterior.
  - Funcionalidad completa
- drmaa_set_vector_attribute:
  - Posible cambio del código de error devuelto: En lugar de DRMAA_ERRNO_INVALID_JOB, DRMAA_ERRNO_INVALID_ARGUMENT, por la misma razón que en el caso anterior.
  - Funcionalidad completa
- drmaa_run_job:
  - Posible cambio del código de error devuelto: En lugar de DRMAA_ERRNO_NO_MEMORY, DRMAA_ERRNO_TRY_LATER, porque a lo mejor es posible intentarlo más tarde cuando alguno de los jobs actuales finalice y sea posible enviar otro.
  - Funcionalidad completa
- drmaa_control:
  - Funcionalidad: En la especificación se indica que cuando el parámetro *jobid* es igual a DRMAA_JOB_IDS_SESSION_ALL hay que aplicar el estado indicado por *action* a todos los jobs emitidos durante esa sesión. Esto no está implementado.
- drmaa_job_ps:
  - Los códigos de error de la implementación son correctos pero no coinciden con los de la documentación de Gridway.
  - Funcionalidad: La versión actual no soporta determinados estados porque no son compatibles con Gridway. Es posible que se puede hacer alguna modificación para admitir algún estado más.
- drmaa_synchronize:

- Posible cambio del código de error devuelto: En lugar de DRMAA_ERRNO_INTERNAL_ERROR, DRMAA_ERRNO_INVALID_ARGUMENT, porqué este último es más específico.

- Funcionalidad: En la implementación actual el parámetro *timeout* sólo puede ser DRMAA_TIMEOUT_WAIT_FOREVER. Hay que ampliarlo para que admita una cantidad de tiempo cualquiera.

- drmaa_wait:

- Posible cambio del código de error devuelto: En lugar de DRMAA_ERRNO_INTERNAL_ERROR, DRMAA_ERRNO_INVALID_ARGUMENT, porqué este último es más específico.

- Funcionalidad: En la implementación actual el parámetro *timeout* sólo puede ser DRMAA_TIMEOUT_WAIT_FOREVER. Hay que ampliarlo para que admita una cantidad de tiempo cualquiera.

- Funciones no implementadas:

- Funciones de ayuda:

- drmaa_get_next_attr_name
- drama_release_attr_names

- Funciones para construir un template:

- drmaa_get_vector_attribute
- drmaa_get_attribute_names
- drmaa_get_vector_attribute_name

- Funciones para sincronización:

- drmaa_wifexited
- drmaa_wetxitstatus
- drmaa_wifsignaled
- drmaa_wtermsig
- drmaa_wcoredump
- drmaa_wifaborted

- Funciones auxiliares:

- drmaa_get_contact
- drmaa_version
- drmaa_get_DRM_system
- drmaa_get_DRM_implementation

- Nuevas funciones que aparecen en la especificación 1.0 del DRAFT:

    - drmaa_get_num_attr_names
    - drmaa_get_num_attr_values
    - drmaa_get_num_job_ids

- Implementación de la señal HARDKILL

  Al igual que en GridWay, esta señal no estaba todavía contemplada. Al hacer un mayor acercamiento al código DRMAA se estimó que el uso de esta señal era de mucha importancia. Como se apunta en la sección de GridWay, el uso de esta función se puede resumir como la capacidad de finalizar trabajos cuando la máquina remota se ha "colgado" o no responde al envío de señales por parte de la máquina origen. Una vez que se implementó esta señal en GridWay, su implementación en DRMAA fue relativamente sencilla.

- Por último, la lista de directivas que faltan por definir para el desarrollo de los templates: Los puestos en cursiva son opcionales, los demás obligatorios.

    - DRMAA_JOB_CATEGORY
    - DRMAA_JOIN_FILES
    - DRMAA_JS_STATE
    - DRMAA_NATIVE_SPECIFICATION
    - DRMAA_START_TIME
    - DRMAA_V_EMAIL
    - DRMAA_V_ENV
    - *DRMAA_CONTACT_BUFFER*
    - *DRMAA_DRM_SYSTEM_BUFFER*
    - *DRMAA_IMPL_BUFFER*
    - *DRMAA_JOBNAME_BUFFER*
    - *DRMAA_SIGNAL_BUFFER*
    - *DRMAA_DONE*
    - *DRMAA_FAILED*
    - *DRMAA_BLOCK_EMAIL*
    - *DRMAA_DEADLINE_TIME*
    - *DRMAA_DURATION_HLIMIT*
    - *DRMAA_DURATION_SLILMIT*
    - *DRMAA_TRANSFER_FILES*
    - *DRMAA_WCT_HLIMIT*
    - *DRMAA_WCT_SLIMIT*
    - *DRMAA_SUBMISSION_STATE_ACTIVE*
    - *DRMAA_SUBMISSION_STATE_HOLD*
    - *DRMAA_PLACEHOLDER_HD*
    - *DRMAA_PLACEHOLDER_WD*
    - *DRMAA_ERRNO_AUTH_FAILURE*

- *DRMAA_ERRNO_NO_DEFAULT_CONTACT_STRING_SELECTED*

Entender la implementación de GridWay necesaria

Éste paso fue quizá el que más tiempo y esfuerzo nos ha llevado. Consistió en sumergirnos en el código de GridWay y entender a la perfección su funcionamiento, y todas las formas de comunicación entre sus diferentes módulos (envío de señales, sistemas de colas de peticiones...)

Nos centramos principalmente en la comprensión de dos módulos, el Request Manager y el Dispatch Manager, aunque posteriormente para implementar alguna señal tuvimos que conocer, aunque con mucha menos profundidad, el funcionamiento del Submission Manager.

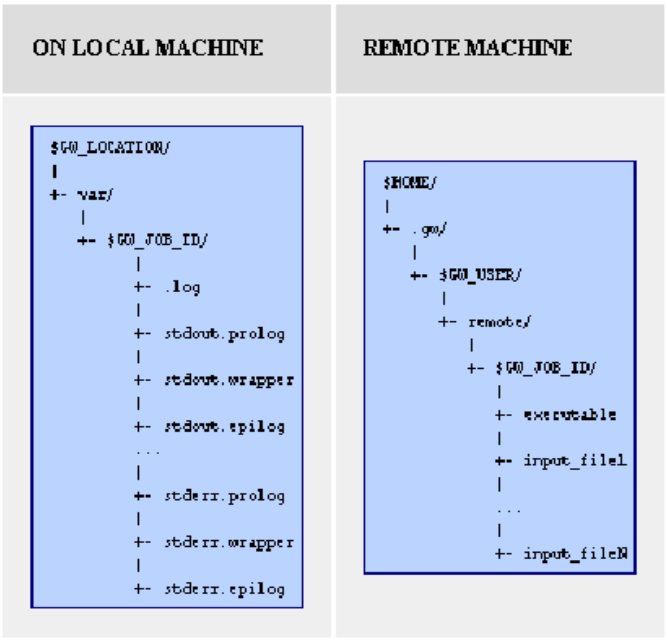
GridWay permite tratar a los jobs como si fuesen procesos Unix. Cada job tiene asignado un identificador único *(JID : Job Identifier)*. Si el job es un array, dispondrá también de un identificador de array *(AID: Array Identifier)* . Las arrays disponen también de un índice con el que acceder a cada job individual dentro de ella *(TID: Task Identifier).*

Las características con las que se crean los jobs a enviar están descritas en una plantilla almacenada en un archivo. En dicho archivo los usuarios podrán especificar las características y los detalles que deseen que tengan los jobs que envíen. Cada job tendrá un atributo llamado Job Template, que serán donde se carguen dichas características. En el archivo template se podrán especificar entre otras cosas, por ejemplo:

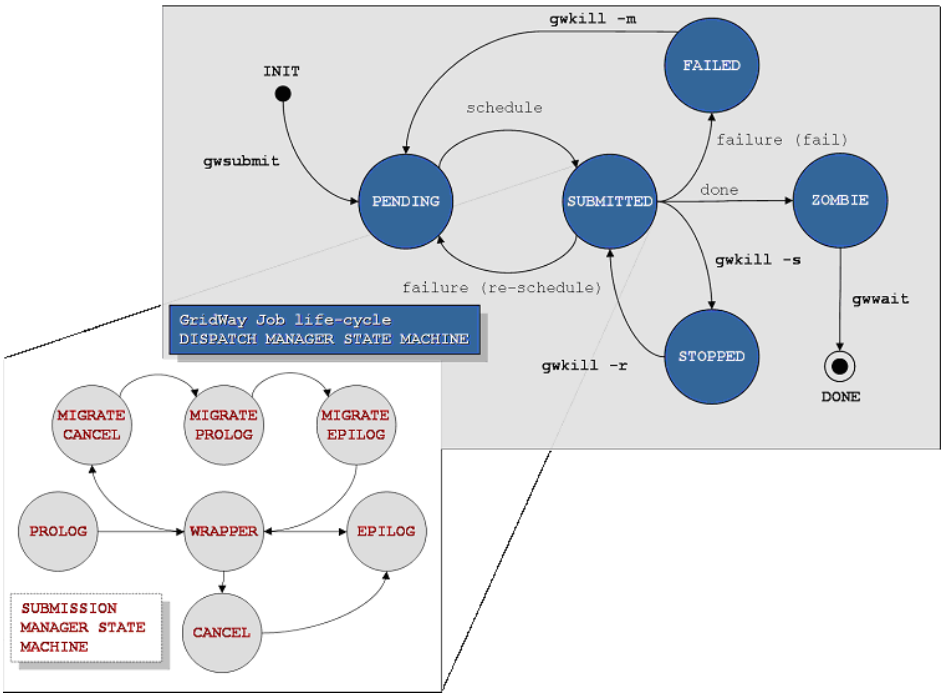
- Rutas : Archivo ejecutable, archivos de entrada, de salida, de error...
- Tiempos: Tiempos de inicio de ejecución, de espera, de parada, de suspensión, de actualización...
- Parámetros : parámetros iniciales, ejecutables de los diferentes módulos...

Para la configuración del job template, se pueden usar algunas de las variables que GridWay define dinámicamente. (como por ejemplo las mencionadas anteriormente).

Cada vez que se envía un job al sistema, el modulo Prolog se encarga de crear una jerarquía de directorios, tanto en el host local como en el remoto donde se guardaran diferentes archivos de información:



Cada job enviado, se encuentra en un estado dentro del sistema, descrito por el siguiente diagrama[2]:



[2] Diagrama de estados inicial, el cual ha sido modificado a lo largo del desarrrollo del proyecto.

Y por último, o quizá debería haber sido lo primero, como ejecutar GridWay:

- Arranque del proxy mediante el comando *"grid-proxy-init"*[3]
- Arranque de GridWay (mediante un demonio) con el comando *"gwd"*[4]
- envío de trabajos mediante el comando *"gwsubmit"*
- envío de señales a los trabajos mediante el comando "*gwkill*"
- Monitorización de trabajos mediante el comando *"gwps"*
- Espera a la finalización de uno o varios jobs mediante el comando "*gwwait*"
- ...

Una explicación más detallada del uso de cada uno de estos comandos se puede obtener en la dirección : http://www.gridway.org/commands.php
Varios ejemplos de uso pueden ser consultados en: http://www.gridway.org/examples.php

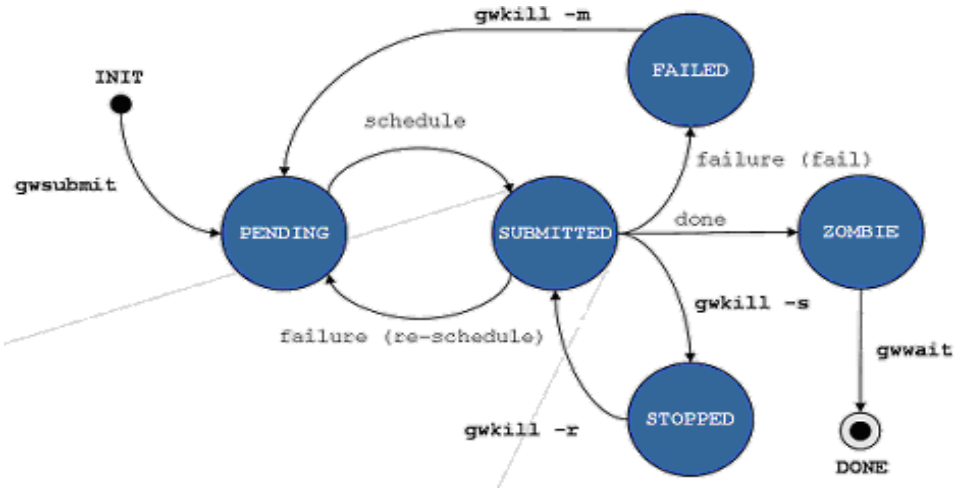
Desarrollo GridWay

El desarrollo de funcionalidad se ha dividido en varias iteraciones, en función de los objetivos planificados en cada reunión de grupo. A continuación se indicará las modificaciones hechas en cada iteración.

Primera iteración

En la primera iteración los cambios que realizados conciernen al estado del job dentro del sistema. En un comienzo, el diagrama de estados por el que pasaba el job a lo largo de su ejecución era:

[3] Tenemos que haber obtenido anteriormente un certificado de autorización para ejecutar GridWay.
[4] Se dispone de un archivo de configuración , denominado gwd.conf , y ubicado en el directorio *$GW_LOCATION*/etc/



Añadimos un nuevo estado , GW_DM_STATE_HOLD. La transición a este estado, se realiza, bien desde el estado GW_DM_PENDING a través del envío de la señal gw_kill correspondiente con el parámetro -h, o bien al enviar el trabajo al sistema inicialmente con la orden gw_submit si se le indica el parámetro –h correspondiente.

El significado de este nuevo estado es que el job no lo tenga en cuenta por el momento el planificador de tareas a la hora de realizar una nueva planificación. En este estado, entrarán los jobs, por ejemplo, cuando posteriormente sea posible poner *Start-time* a los trabajos. O se enviarán los jobs a este estado cuando aunque hayan sido lanzados al sistema, no se quiera que se empiecen a ejecutar aún por diversos motivos. De momento, lo único que hacemos en esta iteración es dotar a GridWay de la funcionalidad necesaria para poder llegar a este estado. Para implementar este nuevo estado, se ha seguido la misma línea que con el resto de señales hay implementadas. Por un lado, se ha definido el nuevo estado. A la hora de lanzar el job inicialmente en estado hold, se ha añadido un parámetro –h al comando gwsubmit. Todas las peticiones de los clientes se envían al request en forma de mensaje ipc, que se almacena en la cola de peticiones del request manager, donde son atendidas en orden. El nuevo parámetro tendrá que ir incluido por tanto dentro del mensaje ipc. Una vez en la cola, cuando el request manager saca el mensaje de la cola para tratarlo y detecta el parámetro hold = 1, envía un nuevo tipo de señal al dispatch Manager, GW_MSG_HOLD, a la que hemos asociado la acción correspondiente para que el job tenga como estado inicial hold, y que será ejecutada cuando el manejador de señales del dispatch Manager reciba la señal.

Para realizar todos estos cambios hemos seguido la siguiente secuencia de acciones:

1. En el archivo gw_job.h:

   - Añadido el estado GW_DM_STATE_HOLD en el enumerado gw_dm_state_t.

   - Creado un nuevo enumerado gw_initial_state_t que tiene como posibles valores pending o hold, que son los dos estados en los que se podrá iniciar un trabajo a partir de ahora[5].

[5] En la version inicial, todos los jobs se inicializaban a pending directamente.

2. En el archivo gw_msg.h:

    - Añadido un nuevo campo en la estructura gw_msg_s que indica el estado inicial y de tipo gw_initial_state_t, denominado on_submit_state.

3. En el archivo gw_submit.c:

    - Añadido el código necesario para que se reconozca la opción -h, que será el parámetro que indique que el job se iniciará en estado HOLD, y se ha tenido que modificar la llamada al método gw_client_job_submit meciéndole un parámetro adicional, en el que se pasa el valor que tiene que tomar el atributo on_submit_state para el mensaje a crear (hold = 1 -> hold, hold = 0 -> pending).

4. En el archivo gw_client.c

    - Modificado el método gw_client_job_submit con el parámetro adicional para fijar el estado inicial a hold o a pending.
    - Añadidas dos nuevas funciones, gw_client_job_hold y gw_client_job_release. Se ejecutan cuando a la señal kill se le ponen los parámetros –h y –l respectivamente, y sirven para pasar del estado pending a hold y viceversa.

5. En el archivo gw_client.h

    - Modificado la cabecera de la función gw_client_job_submit y añadidas las cabeceras de las funciones nuevas.

6. En el archivo gw_kill.c

    - Añadidas dos nuevas opciones, -h y -l para poder pasar un job de pending a hold y viceversa. Para implementar estas dos nuevas opciones hemos tenido que comprobar que fueran compatibles con el resto de opciones.

7. En el archivo gw_rm.c

    - Modificada la función gw_rm_msg añadiéndole nuevas posibilidades para el mensaje (ahora puede tomar dos nuevos valores GW_MSG_HOLD, GW_MSG_RELEASE)

8. En el archivo gw_dm_actions.c

    - En el método dm_submit, es donde realmente se asigna el estado inicial del job en función del campo on_initial_state del mensaje que hemos ido pasando a través de señales hasta este punto.

9. En el archivo gw_dm.c

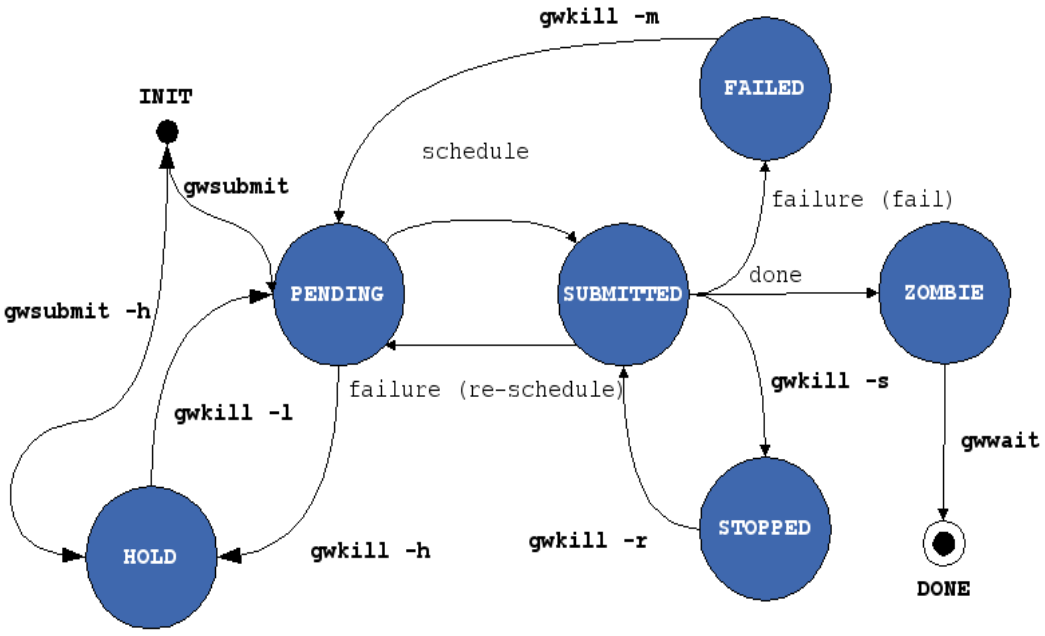
    - Al introducir un nuevo estado, hay que ampliar todas las demás funciones que ya estuviesen implementadas para especificar en cada una de ellas la acción que hay que realizar cuando llegue cualquier señal (kill, wait, sinchronyze...) a un job que se encuentra en estado hold.

- Se registran dos nuevas señales, GW_DM_HOLD y GW_DM_RELEASE, a las que se asocian sus manejadores correspondientes, las funciones gw_dm_hold() y gw_dm_release().
- Además, ya existía la posibilidad de enviar arrays de trabajos, lo cual no es incompatible con que se desee enviar la array en estado hold. Como no es hasta el dispatch donde realmente se crean y se envían los jobs, no se ha tenido en cuenta hasta aquí, pero cuando el mensaje enviado por el RM es atendido y hay que crear los jobs y enviarlos, nos vemos en la necesidad de crear dos nuevas funciones para enviar arrays de trabajos en estado hold. Estas funciones son implementadas y se denominan gw_dm_array_hold y gw_dm_array_release.

10. En el gw_dm.h

    - En este archivo simplemente se han declarado las cabeceras de los métodos implementados

La introducción de GW_DM_STATE_HOLD dejaría el nuevo diagrama de estados del sistema de la siguiente forma:



Segunda iteración

La segunda fase de ampliación de GridWay fue dotar al sistema de una nueva señal, la señal Hard Kill, que es apropiada para aquellos casos en los que el sistema remoto se ha colgado o no responde a las señales que le podamos mandar desde la maquina origen.

En cualquier estado si se recibe esta señal, se enviará una señal de cancel al EM y se producirá la transición al estado DONE del Submission Manager sin espera de respuesta por parte del host remoto (recordemos que puede estar caído). Se ha seguido la misma secuencia de acciones que para el envío de cualquier otra señal, registrando claro está la nueva señal en cada uno de los *listener* de los diferentes módulos con su función manejadora correspondiente, hasta llegar al Submission Manager, donde el manejador de la señal lo que hace es enviar una señal de cancel al EM y poner el estado del job a *zombie*. La razón de por qué llevar al job a este estado y no directamente al estado done, es porque si lo llevásemos a done, al enviar el mensaje de retorno que iría desde el dispatch hasta el request, cuando en el request se tratase, y viese que la tarea ha terminado, eliminaría el job y liberaría la memoria pertinente como se realiza cada vez que un job finaliza. Si después de realizar esto, el host remoto al que se ha enviado la señal hardkill, por lo que fueral, enviase algún tipo de señal de retorno, ésta seguiría el mismo camino EM->SM->DM->RM que normalmente, y en cualquier punto de ese camino, se podría intentar acceder al job al que se refería la señal, dando entonces una violación de acceso a memoria por intentar acceder a algo que ya realmente no existe, porque fue liberado con anterioridad.

La secuencia de acciones que deben realizarse para implementar esta nueva señal, serían por tanto:

- Introducir un nuevo parámetro en la sentencia gwkill para enviar una señal de tipo hardkill.

- Capturar ese parámetro en gw_client_job_kill y pasar un nuevo tipo de mensaje GW_MSG_HARD_KILL al Request Manager.

- Capturar el nuevo tipo de mensaje en el request manager y enviar una nueva señal al Dispatch manager.

- Registrar nuevas acciones en el dispatch manager y en el submission manager.

- El manejador del submission manager modificará el código de estado del SM, poniéndolo a GW_SM_STATE_DONE como si el job se hubiera completado, y del DM, llevándolo a estado zombie, para que no lo tenga en cuenta en las planificaciones pero tampoco lo elimine de momento. Además se envía una señal CANCEL al Execution Manager y se activa el camino de vuelta del mensaje mediante el envío de la señal GW_SM_DONE

Las modificaciones realizadas en cada modulo para implementar esta señal, fueron las siguientes:

1. En el archivo gw_sm_actions.h

    - Añadida la cabecera de las nuevas funciones.

2. En el archivo gw_job.h

   - Añadido un nuevo estado llamado "GW_EXIT_STATUS_HARD_KILLED" en el enum "gw_exit_status_t".

3. En el archivo gw_sm.c:

   - Se ha registrado la acción "GW_SM_HARD_KILL"

4. En el archivo gw_sm_actions.c:

   - Se ha implementado el manejador de esta acción (gw_sm_hard_kill). Se modifica los estados de ejecución de los módulos Dispatch, dm_state, pasándolo *a zombie*, y del Submission, sm_state, pasándolo a *done*. y se envía una señal un *cancel* al EM. Se envía una señal gw_sm_done desde el submission para comenzar el viaje de retorno de la señal hardkill(SM-->DM-->RM), para informar de que se ha llevado a cabo satisfactoriamente.

   - Se ha modificado la función gw_sm_done, para tener en cuenta el nuevo caso.

5. En el archivo gw_kill.c:

   - Se añade el parámetro -9 para enviar esta señal.

6. En el archivo gw_client.c:

   - Añadidos los métodos gw_client_job_kill.c y gw_client_array_kill.c para tratar el parámetro -9, poniendo el tipo de mensaje adecuado, bien GW_MSG_KILL o GW_MSG_HARD_KILL según convenga.

7. En el archivo gw_rm.c:

   - Modificado el método gw_rm_msg, para enviar el mensaje apropiado al dispatch manager.

8. En el archivo gw_dm.c

   - Se ha registrado la acción "GW_DM_HARD_KILL"

9. En el archivo gw_dm_actions.c:

   - Se crea el manejador de la señal gw_dm_hard_kill.

   - Dentro del método gw_dm_done, se añade un case más al switch para tratar el caso en el que el código de salida del job sea hardkill, en el cual, se pasa a zombie el estado dm_state, y se inicia el camino de retorno hacia el Request Manager.

Los usuarios de GridWay tendrán la posibilidad de lanzar este nuevo tipo de señal mediante el parámetro –9 del comando gw_kill, además del resto de opciones que ofrece este comando.

Tercera Iteración

En esta iteración la ampliación de GridWay fue relacionada con todo lo referente a tiempos de configuración. Se amplió la configuración del job Template para poder configurar los trabajos bien con un tiempo de espera máximo, con un tiempo máximo en estado ejecución, con un tiempo total de ejecución, etc.. Además se implementa el atributo StartTime, que permite configurar el job, a través del archivo de configuración que se carga en el job template, con una fecha inicial de ejecución. Cuando un job se crea con un valor para el atributo Start Time distinto de 0, el estado inicialmente se pondrá en estado hold, y no pasará a formar parte del conjunto de trabajos a planificar hasta que no llegue esta fecha de inicio. El planificador tendrá que revisar en cada iteración todos los jobs que estén a hold y mirar su fecha de inicio, para realizar las acciones oportunas. Todos estos tiempos se añaden como atributos en el template con el que se configuran todos los trabajos. Los pasos a seguir para dotar al template de cada uno de estos atributos son los mismos para todos, y consisten básicamente en:

- Primeramente, se han añadido los diferentes atributos a la estructura gw_job_template_s.

- A continuación se tiene en cuenta los posibles nuevos atributos que pueden aparecer en el archivo de configuración a la hora de leer éste, rellenando correctamente los campos del template.

- A la hora de gestionar los trabajos, el planificador tendrá que tener en cuenta todos estos nuevos tiempos, para finalizar o comenzar trabajos en función de los valores que cada uno de estos limites pueda marcar.

A continuación, se pasa a explicar de una forma más detallada cada uno de los atributos añadidos al template, así como su funcionalidad exacta y lo que hemos hecho para implementarlo.

Atributo Start Time

Dotar al sistema de esta funcionalidad, da una gran flexibilidad a los programadores o usuarios de GridWay a la hora de enviar trabajos y gestionar su ejecución, ya que permite el envío de éstos al sistema pero que no comiencen a ejecutarse hasta un determinado tiempo después.

Por un lado, hemos tenido que modificar la estructura job y la estructura job_template para añadir el atributo start time de tipo time_t. Este atributo será el que indique la fecha de comienzo de la ejecución del job, o mejor dicho, el momento en el que el job pasa a formar parte del conjunto de trabajos a planificar.

El valor de este atributo se asigna dentro del archivo de configuración correspondiente, en el que se ha añadido una nueva línea:

START_TIME = 2005/04/19-14:40:00

Para que la fecha indicada en el archivo de configuración sea reconocida, debe tener el siguiente formato:

"[YY]YY/MM/DD-HH:MI:SS"

Una vez que disponemos del atributo, pasamos a indicar los pasos para reconocer dicho valor:

- Inicialmente, el atributo start_time se inicializa a 0 cuando se crea el job dentro de gw_job_init.
- posteriormente se asigna el valor dentro del método gw_job_template_fill en el que se lee el valor del atributo si es que se ha asignado un valor para él.
- Por último, se usa dicho valor para planificar adecuadamente el job en el método gw_dm_schedule

A continuación detallamos los archivos modificados:

1. En el archivo gw_job.c
    - El método gw_job_init se inicializa el atributo a 0
2. En el archivo gw_job_template.c
    - Se ha añadido el código necesario dentro de gw_job_template_fill para rellenar el nuevo parámetro.
3. En el archivo gw_dm_actions.c
    - Cuando se hace un submit de un job con un valor distinto de cero para el atributo startTime, el job se inicializa en estado hold.
4. En el archivo gw_dm_schedule
    - Cuando se realiza el reschedule comprueba si los trabajos en hold tienen que cambiar a pending porque se haya alcanzado ya el tiempo de start time indicado en su template.

Atributo duration_hlimit

Este atributo especifica qué cantidad de tiempo el job debe estar en estado de ejecución antes de que su limite se haya excedido y por ello el sistema DRMS decida finalizarlo.

Para conseguir esta acción, dentro del método gw_dm_schedule se compara el tiempo duration_hlimit que aparece en la template del job con el tiempo wrapper_time del job. Si el primero excede al segundo, se manda una señal kill para eliminar dicho job.

Atributo duration_slimit

Este atributo especifica una estimación de que tiempo sería necesario para que el job pase de estado running a complete.

Este atributo puede ser útil para el planificador, ya que en base a esos valores y a la política de planificación que tenga determinada, se ejecutara un job u otro. Si el tiempo especificado es insuficiente, la implementación drmaa debería imponer una penalización.

Atributo deadline

El atributo deadline especifica una fecha tope después de la cual el sistema DRMS hará finalizar el job aunque este no haya completado su ejecución.

Atributo wct_hlimit

Este atributo especifica un "*tiempo de pared*", que indica el tiempo que está permitido que un job consuma antes de que este limite se haya excedido. En este tiempo no sólo se tiene en cuenta el tiempo en ejecución, sino todo el tiempo desde que el job se lanza, incluido el tiempo de suspensión.

Cuando un job excede su tiempo wct_hlimit, el sistema DRMS deberá terminar el job.

Para conseguir esta acción el método gw_dm_schedule compara el atributo wct_hlimit del template del job con el tiempo total que lleva el job activo (tiempo actual – start time)

Si el primero ha superado esta cantidad se envía una señal kill para eliminar el job.

Atributo wct_slimit

Este atributo especifica una estimación del tiempo de pared (wct_hlimit) que un job necesitaría para completarse. Este atributo puede resultar útil al planificador, porque en función del tiempo que se estime necesario para un job u otro, se podrá decidir que job ejecutar primero.

Cuarta iteración

En esta iteración, los cambios realizados han sido para aumentar la funcionalidad tanto de las señales wait como synchronize

Implementación de gw_job_wait con timeout

La implementación de una señal gwwait con timeout es de gran importancia, ya que permite que numerosos programas realicen una planificación más eficiente de sus tareas, descartando esperas indefinidas, o implantando un plazo máximo de expiración de trabajos.

Para realizar la implementación de esta funcionalidad en primer lugar tuvimos que aprender cual es el proceso que realizaba GridWay para realizar un wait:

1. Se realiza una llamada a la función gw_client_job_wait. A no ser que este activada la opción DRMAA_JOB_IDS_SESSION_ANY en cuyo caso se debe esperar a que cualquier trabajo de la sesión termine. El proceso es bastante similar.

2. gw_client_job_wait envía un mensaje del tipo GW_MSG_WAIT al Request Manager

3. El gestor de eventos del Request Manager detecta la presencia del mensaje y se dispara un evento dm_wait en el Dispatch Manager.

4. El gestor de eventos del Dispatch Manager chequea si el trabajo ha finalizado y si ese es el caso devuelve un mensaje(realmente dispara un evento en el Request Manager y es este él que envía el mensaje), incluyendo el código de retorno.

5. Si no ha terminado, se marca el trabajo (atributo job.waiting==GW_TRUE).

6. Cuando el trabajo termina, de forma normal o por señal, si tiene el flag waiting a cierto entonces se activa el evento en el Request Manager y se envía el mensaje correspondiente.

7. Una vez recibido el mensaje se procesa el código de retorno y finaliza la función.

Una vez analizado el problema, decidimos que la mejor solución era realizar la comprobación del *"timeout"* mediante la llamada para enviar el mensaje. Lamentablemente los mensajes IPC, que es el método de comunicación elegido en GridWay para la comunicación entre gw_client y el Request Manager, no permite mensajes con un timeout, sino que se limita a mensajes de dos tipos:

- Espera indefinida
- Sin espera

Por tanto para implementar la funcionalidad requerida nos vimos obligados a realizar una espera activa. Realizamos un bucle comprobado en cada iteración:

1. Si se ha recibido el mensaje de fin de ejecución, mediante mensajes sin espera.

2. Si se ha superado el timeout fijado.

Una vez que se alcanza cualquiera de estas dos opciones se finaliza la función devolviendo el código correspondiente.

Aunque no es la solución más limpia deseable, debido al sistema de paso de mensajes actual de GridWay , que es posible cambie en futuras versiones, ha sido la única forma que hemos considerado factible.

Otra posible solución hubiera podido ser, por ejemplo, la activación de un flag en Dispatch Manager, que comprobase durante la planificación de tareas si se ha superado el tiempo de wait. Pero esta opción fue descartada inmediatamente. En primer lugar porque debe ser el cliente el que decide cuanto tiempo ha de esperar y en que momento corta la comunicación, y no olvidemos que la planificación sólo se activa cada cierto periodo de tiempo. Y en segundo lugar, no podemos sobrecargar la tarea del planificador, ya costosa de por si, con comprobaciones que no son de su competencia.

Para que los clientes de GridWay puedan utilizar esta nueva versión de gw_wait, añadimos una nueva opción, -t seguida del tiempo de espera máximo en segundos. De todos modos, también sigue disponible la versión anterior de gw_wait en la que se realiza una espera indefinida.

Los archivos modificados han sido los siguientes:

1. En el archivo gw_client.c:
   - Se añade un nuevo parámetro timeout a los siguientes métodos:
     - gw_client_job_wait
     - gw_client_wait_array
     - gw_client_job_wait_set
     - gw_client_job_wait_any_set
   - En gw_client_job_wait, después de enviar el mensaje se calcula el tiempo actual y se le suma este parámetro. Éste será el tiempo final tras el cual el trabajo deberá ser finalizado. Si la llamada a la función se hace con un tiempo igual a 0 se considerará que se trata de un wait normal (sin tiempo). Si se supera el timeout se procederá a devolver el código GW_RC_TIMEOUT.

2. En el archivo gw.common.h
   - Se añade un nuevo tipo de código de retorno: GW_RC_TIMEOUT para indicar que el wait ha agotado su tiempo de espera.

El uso de esta nueva funcionalidad desde DRMAA se utilizará tanto en la implementación de del gw_wait como del gw_synchronize con “timeout”.

Desarrollo DRMAA

Como se ha explicado anteriormente en la sección *Revisar la implementación inicial de DRMAA en GridWay,* se destacaron una serie de posibles modificaciones de la versión inicial sobre las que trabajar. A continuación se detallan las soluciones aportadas en cada iteración de trabajo.

Primera iteración

En primer lugar, se cambiaron los códigos de error que se estimaron incorrectos en la revisión de la implementación. Las funciones para las que se cambiaron estos códigos son las siguientes:

- drmaa_get_next_attr_value: DRMAA_NO_MORE_ELEMENTS en lugar de DRMAA_ERRNO_INTERNAL_ERROR.

- drmaa_get_next_job_id: DRMAA_NO_MORE_ELEMENTS en lugar de DRMAA_ERRNO_INTERNAL_ERROR,

- drmaa_init: DRMAA_ERRNO_INVALID_ARGUMENT en lugar de DRMAA_ERRNO_INVALID_CONTACT_STRING, y DRMAA_ERRNO_DRMS_INIT_FAILED en lugar de DRMAA_ERRNO_DRM_COMMUNICATION_FAILURE,

- drmaa_exit: Cuando no hay ninguna sesión activa pasa a devolver DRMAA_ERRNO_NO_ACTIVE_SESSION en lugar de DRMAA_SUCCESS.

- drmaa_delete_job_template: DRMAA_ERRNO_INVALID_ARGUMENT en lugar de
  DRMAA_ERRNO_INVALID_JOB.

- drmaa_set_attribute: DRMAA_ERRNO_INVALID_ARGUMENT en lugar de DRMAA_ERRNO_INVALID_JOB,
  DRMAA_ERRNO_INVALID_ARGUMENT.

- drmaa_get_attribute: DRMAA_ERRNO_INVALID_ARGUMENT en lugar de DRMAA_ERRNO_INVALID_JOB,
  DRMAA_ERRNO_INVALID_ARGUMENT.

- drmaa_set_vector_attribute: DRMAA_ERRNO_INVALID_ARGUMENT en lugar de DRMAA_ERRNO_INVALID_JOB,
  DRMAA_ERRNO_INVALID_ARGUMENT.

- drmaa_run_job: DRMAA_ERRNO_TRY_LATER en lugar de DRMAA_ERRNO_NO_MEMORY.

- drmaa_synchronize: DRMAA_ERRNO_INVALID_ARGUMENT en lugar de DRMAA_ERRNO_INTERNAL_ERROR.

- drmaa_wait: DRMAA_ERRNO_INVALID_ARGUMENT en lugar de DRMAA_ERRNO_INTERNAL_ERROR.

Segunda iteración

En esta iteración se procedió a la parte más importante del desarrollo del proyecto en lo relativo al DRMAA. Esto es, la ampliación de la funcionalidad de determinadas funciones. Algunas de las funciones más importantes son *drmaa_wait y drmaa_synchronize*.

Aumento de Funcionalidad de la instrucción drmaa_wait

Como hemos dicho anteriormente, no todas las funciones implementadas de DRMAA gozaban de total funcionalidad tal y como exige la especificación del Global Grid Forum. En el caso de la función drmaa_wait, que recordamos tiene el siguiente aspecto:

```
int drmaa_wait(job_id,int stat, signed long timeout,string array
rusage,drmaa_context_error_buf)
```

La falta de funcionalidad se da en la imposibilidad de usar el argumento de entrada timeout. En la versión implementada de DRMAA para GridWay sólo se permitía la opción DRMAA_WAIT_FOREVER inicialmente.

Esto supone una gran deficiencia, impidiendo, por ejemplo, la programación de aplicaciones que consulten, de manera periódica, si un programa ha terminado y, si no fuera el caso, continuasen realizando otro tipo de tareas. Lo que puede resultar una importante ganancia de tiempo al paralelizar cálculos locales con los trabajos remotos.

Para poder completar la implementación de esta función DRMAA, primero deberíamos dar soporte en GridWay, ya que la versión actual de este no disponía de los atributos ni las funciones necesarias para atender un wait de este tipo. Los cambios que se realizaron en GridWay, ya han sido comentados en el apartado *"Desarrollo GridWay"*. Suponiendo por tanto, que partimos de una base ya implementada en GridWay, completar la funcionalidad del wait en DRMAA es una tarea simple, ya que únicamente hay que modificarla para que en lugar de lanzar un error cuando el parámetro timeout fuese distinto de cero, ahora lo que tendrá que realizarse es una llamada a la función correspondiente de GridWay, en este caso gw_wait pasándole como parámetro el valor correspondiente indicado.

Aumento de funcionalidad de la instrucción drmaa_synchronize

Esta función como la anterior no estaba totalmente implementada, y al igual que la función drmaa_wait su limitación residía en la falta de implementación del tiempo de espera máximo.

Puesto que la función drmaa_synchronize depende indirectamente de la función gw_wait, el análisis y el cambio fueron más rápidos y sencillos.

En primer lugar veremos como funcionaba la instrucción drmaa_synchronize antes de su aumento de funcionalidad:

- Se calculaba el conjunto de trabajos que debían sincronizarse. Existen dos opciones:
  - Todas los trabajos de la sesión DRMAA
  - Un array de identificadores de trabajo
- Se realiza una llamada gw_client_job_wait_set con el conjunto final de trabajos a sincronizar.
- Dentro de gw_client_job_wait_set se comprueba uno a uno que todos los trabajos han terminado mediante un bucle **for** que recorre todo el conjunto de jobs y lanza un gw_client_job_wait por cada job.
- Se devuelven un array con todos los códigos de retorno.

Para aprovechar el código ya creado en gw_client_job_wait, nuestra solución para implementar el "timeout" es:
Dentro de la función gw_client_job_wait_set, recorremos mediante un bucle **for** todo el conjunto de trabajos que se van a sincronizarse, exactamente como se hacía hasta ahora, pero calculando a cada iteración el timeout restante.

Aunque en un principio pudiera considerarse la posibilidad utilizar un bucle **while** para mejorar el rendimiento, nuestra idea es que la perdida de tiempo una vez alcanzado el timeout es corta, cada llamada con timeout cero se resolverá rápidamente y a cambio obtendremos los códigos de salida, en el caso de haber finalizado, de todos los trabajos involucrados en la llamada, de ahí que optáramos por esta solución.

Aumento de la funcionalidad de otras funciones

- drmaa_control:

  La función drmaa_control tiene como finalidad detener, reaudar o matar el trabajo cuyo identificador está especificado como parámetro (Si este identificador es DRMAA_JOB_IDS_SESSION_ALL entonces esta rutina actúa sobre todos los trabajos enviados durante esta sesión).
  El cambio más significativo que se realizó en esta función fue la gestión de la señal HARDKILL. Cuando se comprueba la acción de entrada, hay una señal adicional que no estaba contemplada y, en consecuencia, no era tratada inicialmente. Esta señal es DRMAA_CONTROL_HARD_TERMINATE. Cuando se recibe esta señal, la función drmaa_control realiza la llamada oportuna a gw_client_kill con el parámetro correspondiente a *9*, para que así GridWay realice el tratamiento oportuno.

  También se amplió la funcionalidad de esta función permitiéndola enviar señales de tipo hold y release. Al detectarse este valor en el parámetro,

DRMAA_CONTROL_HOLD o DRMAA_CONTROL_RELEASE. de entrada se procede a llamar a las función gw_client_kill con los parámetros necesarios en cada caso. Pasando los trabajos desde el estado pending al estado hold(DRMAA_CONTROL_HOLD) o del estado hold al estado pending (DRMAA_CONTROL_RELEASE)

- drmaa_job_ps:

  Esta función devuelve el estado del trabajo cuyo identificador es el especificado como parámetro.

  La ampliación de esta rutina se debió a la introducción en GridWay del nuevo estado GW_DM_STATE_HOLD. Debido a la falta inicial de este estado, el estado DRMAA_PS_USER_ON_HOLD de DRMAA (equivalente al anterior en GridWay) estaba definido pero no implementado. Por tanto, una vez introducido e implementado en GridWay, su gestión en DRMAA fue más sencilla.

  En la función ps se introdujo un nuevo case para atender al estado hold

  ```
  case GW_DM_STATE_HOLD
      remote_ps = DRMAA_PS_USER_ON_HOLD
  break;
  ```

Tercera iteración

En esta iteración se procedió a la implementación de la señal HARDKILL y a la gestión de hilos. A continuación se detallan ambos puntos.

Implementación de la señal HARDKILL

Como ya se apuntó anteriormente en la sección de adquisición de conocimiento y más concretamente en la revisión de (.....), la implementación de esta señal en DRMAA estaba supeditada a su implementación en GridWay. Así, una vez hecho en GridWay, las modificaciones introducidas en DRMAA con el fin de introducir esta señal fueron las siguientes:

- Se registra la señal proporcionándole un nuevo código (esto se realiza en el archivo gw_drmaa.h

  #define DRMAA_CONTROL_HARD_TERMINATE 5

- drmaa_control: Cuando se comprueba la acción, si ésta es DRMAA_CONTROL_TERMINATE, se llama al método gw_client_job_kill pasándole un 0 como parámetro. En el caso de que sea DRMAA_CONTROL_HARD_TERMINATE, en lugar de pasarle un 0 se le pasa un 9. Será GridWay el que gestione las operaciones oportunas según el caso.

Implementación del DRMAA THREAD-SAFE

Los autores de la especificación DRMAA esperan que los desarrolladores utilicen la librería DRMAA tanto con códigos secuenciales como *multithreaded*. Por tanto recomiendan que las implementaciones de la librería sean **thread-safe,** y que permita a las aplicaciones *multithreaded* el uso de las instrucciones DRMAA sin que sea necesario ninguna sincronización explicita entre los hilos de la aplicación, y por supuesto, que funcione correctamente.

Por otra parte exigen que la rutina de inicialización de sesión (drmaa_init), y la de finalización de sesión, sean ejecutadas por un único hilo, el hilo maestro en la mayoría de los casos.

Por ultimo también se recomienda que la implementación sea realizada mediante estándares, como POSIX, en caso de realizar otra implementación esta deberá estar debidamente documentada.

Al inicio del proyecto la implementación DRMAA desarrollada no era *thread-safe*, por lo que se decidió dotarla de mecanismos de sincronización para permitir la concurrencia de hilos, usando la librería DRMAA, tal y como se recomienda en la especificación 1.0 del Global Grid Forum.

Para realizar la sincronización en primer lugar estudiamos, que recursos eran solicitados en las regiones criticas y cuales eran dichas regiones. Los recursos compartidos que localizamos fueron:

- Estructura de datos drmaa_gw_session_s: Almacena los datos de la sesión(pid del proceso) y los identificadores trabajos lanzados durante la misma.

- Estructura de datos drmaa_gw_job_ids_s: Sirve para almacenar un conjunto de identificadores de trabajo.

- Estructura de datos drmaa_attr_values_s: Almacena los valores de un conjunto de atributos.

- Estructura drmaa_job_template_s: Almacena los atributos y sus valores de un trabajo.

Para garantizar la exclusión mutua, utilizamos semáforos POSIX. Elegimos esta opción tras estudiar el código de las funciones de la librería DRMAA. En la mayor parte de los casos solamente es necesario bloquear un elemento para realizar las operaciones, y en los pocos casos en que era necesario el bloqueo de varios elementos estos siempre se podían bloquear siguiendo un orden establecido:

1. drmaa_gw_session_s.
2. drmaa_job_template_s
3. drmaa_gw_job_ids_s
4. drmaa_attr_values_s

Por lo tanto se garantiza la ausencia de interbloqueos.

Cuarta iteración

Por último, en esta iteración se pasó a implementar aquellas funciones que no estaban implementadas. Estas rutinas se pueden agrupar en las secciones que aparecen a continuación, y la funcionalidad de todas ellas está explicada en la sección Descripción general de las rutinas

Funciones de ayuda

- drmaa_get_next_attr_name
- drama_release_attr_names

Funciones para construir un template

- drmaa_get_vector_attribute
- drmaa_get_attribute_names
- drmaa_get_vector_attribute_name

Funciones auxiliares

- drmaa_get_contact
- drmaa_version
- drmaa_get_DRM_system
- drmaa_get_DRM_implementation

Nuevas funciones que aparecen en la especificación 1.0 del DRAFT

- drmaa_get_num_attr_names
- drmaa_get_num_attr_values
- drmaa_get_num_job_ids

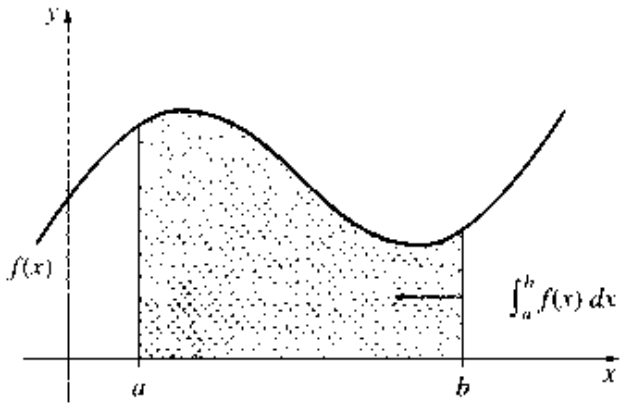
Pruebas realizadas

Banco de pruebas para DRMAA.

Para comprobar las nuevas funcionalidades y cambios que introdujimos en el sistema realizamos un pequeño banco de pruebas, tras la inclusión de cada una de las partes fundamentales de nuestro proyecto. Este banco de pruebas esta fundamentado en el programa drmaa_pi.

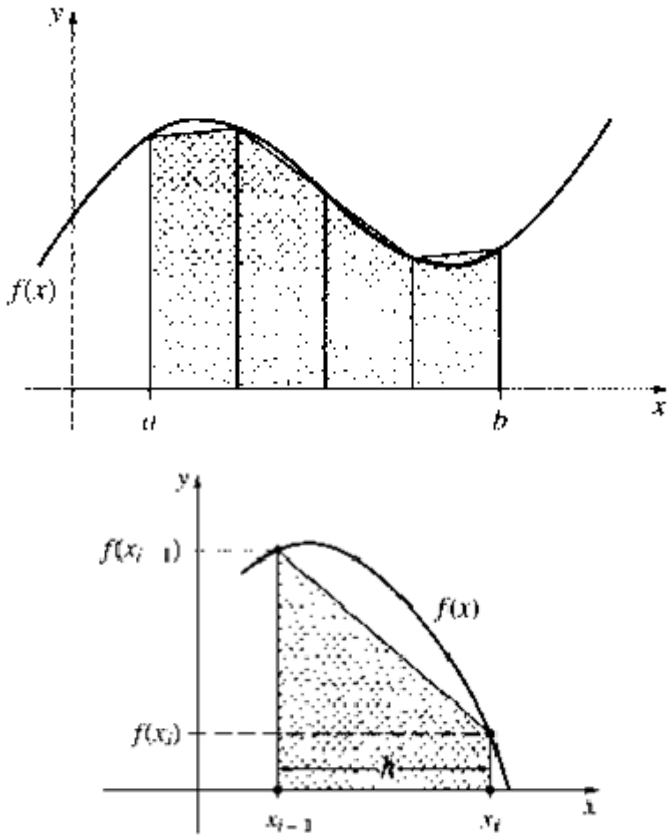
Definición del problema.

El cálculo de π es un problema muy conocido. Para nuestro propósito calcularemos la integral de la siguiente función.



Siendo $f(x) = \frac{4}{1+x^2}$, π será $\int_0^1 \frac{4}{1+x^2} dx$.

Para calcular la totalidad de la integral es interesante dividir la función en numerosas secciones y computarlas cada una por separado.



Como se puede observar, a mayor número de divisiones mejor será la precisión en el cálculo del número π.

Resolución del problema mediante DRMAA para GridWay.

En primer lugar debemos crear el código que calcule el área de cada una de las secciones en las que hemos dividido el programa central. Hemos utilizado C para la creación del código.

```
#include <stdio.h>
#include <string.h>
int main (int nargs, char** args)
{   int rank;
    int total;
    int i, n;
    double l_sum, x, h;
    rank  = atoi(args[1]);
    total = atoi(args[2]);
    n     = atoi(args[3]);
    h = 1.0/n;
    l_sum = 0.;
    for (i = rank; i < n; i += total)
    {   x = (i+0.5)*h;
        l_sum += 4.0/(1.0+x*x);
    }
    l_sum *= h;
    printf("%g\n",l_sum);
    return 0;

    }
```

Al ejecutable resultante tras la compilación, le dimos el nombre de pi.

Una vez que tenemos el ejecutable, el código con DRMAA se compone de estos elementos:

- Inicialización de la sesión. Mediante una llamada INIT.
- Creación del Job Template.
- envío del array de trabajos
- Espera hasta la finalización de todos los trabajos.
- Mostrar los resultados.
- Eliminar el template y cerrar Sesión

La creación del Job Template ocupa la mayor parte del código y es en la parte en la que más se centro nuestro banco de pruebas, ya que la mayor parte del proyecto consistió en la incorporación de atributos DRMAA no implementados.

Para la creación del job template usaremos la sentencia drmaa_allocate_job_template. Cada atributo se le añade utilizando sentencias drmaa_set_attribute

Los atributos más importantes que tuvieron todas las implementaciones del programa de pruebas fueron:

DRMAA_V_ARG =DRMAA_GW_TASK_ID,DRMAA_GW_TOTAL_TASKS,10000

Los dos primeros atributos son valores para determinar que tarea se esta ejecutando y cuantas tareas en total se ejecutaran, el tercer argumento sirve para indicar el número de iteraciones que se utilizaran para calcular el área.

DRMAA_REMOTE_COMMAND=”pi”

Como dijimos antes el ejecutable tenía por nombre pi.

DRMAA_OUTPUT_PATH=”strdout.”DRMAA_GW_TASK_ID
DRMAA_ERROR_PATH=”strderr.”DRMAA_GW_TASK_ID

Estos atributos son los más importantes ya que cada ejecución de pi dejará sus resultados en estos ficheros que luego, una vez que todos los trabajos enviados hayan terminado, serán procesados para el cálculo del número π.

Aparte de los atributos DRMAA necesarios debemos añadir para el funcionamiento estos atributos GridWay

GW_HOST_LIST=wd+./host.list. Siendo wd el directorio en el que se encuentra el fichero

Este atributo es fundamental para que GridWay seleccione el servidor adecuado.

Una vez hemos generado el job template, debemos enviarlo y después esperar a que todos los trabajos finalicen. Esto se realiza con dos sentencias:

- drmaa_run_bulk_jobs. Para enviar un array de trabajos al DRM.

- drmaa_synchronize. Con los flags para todos los trabajos de la sesión y tiempo ilimitado de espera activados.

El motivo de tener estos flags es obvio, en la sesión únicamente existen los trabajos que hemos enviado para el cálculo de pi, por otro lado el timeout de espera no se encontraba activado al iniciar este proyecto.

Una vez han acabado todos los trabajos, el programa realiza la suma de las áreas de cada uno de los subprogramas, mostrando los resultados por pantalla, tanto los resultados de cada trabajo como la suma total.

Finalmente se liberan las estructuras como el job template y el drmaa_job_ids, y se cierra la sesión con drmaa_exit.

*Estado **hold** y señales **hold** y **release***

Una vez implementado el estado hold lo sometimos a las siguientes pruebas:

1. Añadimos al programa drmaa_pi la sentencia para que el trabajo comenzara en este estado.

2. Creamos dos programas drmaa_hold y drmaa_release que lanzaban señales al DRM mediante las instrucciones drmaa_control.

Atributo DRMAA_START_TIME

Para las pruebas sobre el funcionamiento de este atributo, modificamos el código de drmaa_pi introduciendo el atributo start-time con diferentes fechas, mediante la sentencia drmaa_set_attribute, para comprobar su funcionamiento. Por otro lado también realizamos pruebas enviando señales hold y release (Una vez que en trabajo recibe una señal hold, se anula la función del atributo start_time).

Atributos WCT_HLIMIT y DURATION_HLIMIT:

Para las pruebas de funcionamiento de estos dos atributos, introdujimos ambos atributos, tanto simultáneamente como por separado, en el job template mediante sentencias drmaa_set_attribute. Para alterar los tiempos de duración del ejecutable pi, demasiado cortos para que el sistema pudiese chequear los tiempos; GridWay, por defecto, chequea los trabajos cada 20 segundo, mientras que el tiempo de ejecución de pi no suele exceder de los 15 segundos.

Señal hard-kill

Las pruebas de la señal Hard-Kill las hicimos directamente sobre GridWay. Puesto que en el estándar DRMAA no esta presente nada parecido. Pero entre los diversos trabajos que utilizamos como muestra a la hora de lanzar este tipo de señales, si estaban presentes ejecuciones lanzadas mediante DRMAA.

Wait y sinchronize con timeout

Para realizar estas pruebas introdujimos diversos valores de timeout para cada una de las dos instrucciones. En particular las instrucción wait de DRMAA no la probamos, sino que utilizamos en su lugar directamente la función gw_wait, ambas utilizan el mismo método para realizar la espera por lo que los resultados de gw_wait fueron lo bastante ilustrativos para poder comprobar el correcto funcionamiento.

Thread-Safe

La ultima parte del proyecto consistió en intentar convertir la librería DRMAA para que admitiera multi-threading, sin necesidad explicita de sincronismo entre hilos. Para ello realizamos una variante del cálculo del pi que funciona de la siguiente forma:

1. Creamos un hilo por cada tarea pi que deseemos ejecutar.

2. En cada hilo generamos un job template(drmaa_allocate_job_template) con sus atributos (drmaa_set_attribute), y lo ejecutamos (drmaa_run_job) y esperamos a que termine su job(drmaa_wait)

3. Cuando todos los hilos han terminado se computa el resultado y se muestra.

El principal problemas es, que aunque creemos que la librería DRMAA si permite la ejecución no explícitamente sincronizada de hilos, (las estructuras de datos son pocas y siempre se usan en un mismo orden), GridWay no soporta multi-threading, todavía. En particular hemos descubierto problemas en el identificador de mensajes, dos threads realizando la misma operación envían mensajes IPC idénticos, lo que hace que operaciones concurrentes de wait puedan leer mensajes no destinados a ellas y bloquear la aplicación.

Bibliografía

Para la realización de este proyecto se ha empleado información proveniente de dos tipos de fuentes:

- Páginas web:
  - Página del centro de difusión de tecnologías ETSIT – UPM

    http://www.ceditec.etsit.upm.es/grid_computing.php
  - Distributed Resource Management Application API Working Group (DRMAA-WG)

    http://www.drmaa.org/
  - Página de GridForge

    https://forge.gridforum.org/
    http://www.ggf.org/
  - Sección de artículos de

    http://barrapunto.com
  - Sitio oficial de GridWay

    http://www.gridway.org/

Agradecimientos

Queremos mencionar especialmente como muestra de agradecimiento a las siguientes personas que nos han ayudado en el desarrollo de nuestro proyecto, sacándonos de mas de un apuro:

Ignacio Martín Llorente
Profesor de Departamento de Arquitectura de Computadores y Automática
Profesor director del proyecto

Rubén Santiago Montero
Profesor de Departamento de Arquitectura de Computadores y Automática

Jose Luis Vázquez
Colaborador del Departamento de Arquitectura de Computadores y Automática

Autorización

El presente documento sirve para mostrar la total conformidad de los tres componentes del grupo de Sistemas Informáticos del curso 2004-2005 que ha desarrollado el proyecto fin de carrera denominado "*Implementación del Estándar DRMAA para enviar, gestionar y terminar trabajos en Grid*" en **autorizar a la Universidad Complutense para difundir y utilizar con fines académicos, no comerciales, y mencionando expresamente a sus autores, tanto esta memoria como el código, la documentación y/o el prototipo desarrollado**.

Para garantizar dicha autorización, a continuación sirva la identificación y firma de cada uno de los componentes de este grupo de trabajo:

Sergio Espartero Díaz
DNI Nº: 51 984 180 V
Firma:

Beatriz Palazuelos Moya
DNI Nº: 50 880 979 B
Firma:

Patricia Arroyo Peceño
DNI Nº: 52 881 559 M
Firma:

Anexo: drmaa.h